Degree Companion

Christian Burtscher, Jonas Glaser, Marcus Gugacs, and Eva Haring

706.041 Information Architecture and Web Usability 3VU WS 2025/2026
Graz University of Technology

02 Feb 2026

Abstract

This report presents Degree Companion, a web-based degree progress manager, designed to help university students plan their academic journey. The application is built in Svelte and uses Tauri 2.0 for cross-platform deployment. It addresses common challenges students face when navigating complex curricula with numerous modules, prerequisites, and interdependencies. Many students struggle to maintain a clear overview of their progress and often encounter difficulties selecting courses in the correct order.

By providing tools to track completed courses, visualize progress, and plan future semesters while respecting course dependencies, Degree Companion aims to reduce stress and ensure a structured learning path. The project was developed as part of a university course with the goal of creating a practical and user-friendly solution for academic planning. This report covers the project's motivation and context, provides a detailed description of the technical implementation, including the frameworks and libraries used, explains the setup and deployment process, and outlines potential future enhancements to the application.

Contents

Contents i

List of Figures iii

List of Listings v

1 Introduction 1

2 Setup and Build Procedure 3

3 Implementation 5

3.1 Technology Stack . . . 5

3.1.1 Frontend Framework . . . 5

3.1.2 Desktop Integration . . . 5

3.1.3 Visualization and Data Processing . . . 6

3.2 Application Architecture . . . 6

3.2.1 State Management . . . 7

3.2.2 State Persistence and Synchronization . . . 7

3.2.3 State Recovery and Data Loading . . . 8

3.2.4 State Reset . . . 8

3.3 Application Workflow . . . 8

3.3.1 Initial Loading and State Detection . . . 9

3.3.2 Data Import Process . . . 9

3.3.3 Onboarding and Configuration . . . 11

3.3.4 Graph Visualization and Interaction . . . 12

3.4 Component Architecture . . . 13

3.4.1 Component Organization . . . 13

3.4.2 Custom Node Implementation. . . . 14

3.4.3 Edge Rendering and Prerequisites . . . 14

3.4.4 Reactive Data Flow . . . 14

4 Future Work 15

5 Concluding Remarks 17

Bibliography 19

List of Figures

3.1 Degree Companion: Data Import with Table Preview 11
3.2 Degree Companion: User Onboarding Interface 12
3.3 Degree Companion: Curriculum Graph 13

List of Listings

2.1 Modifications to Tauri Workflow 4

3.1 Application State Stores 7
3.2 Storage Manager 8
3.3 State Recovery Mechanism 9
3.4 State Reset Function 10
3.5 Checking for Saved State on Initialization 10

Chapter 1

Introduction

University students today are granted significant freedom in shaping their academic paths. While this flexibility allows personalized learning experiences, it also introduces considerable complexity into degree planning. Students often struggle to navigate the extensive catalog of available modules and courses, each with its set of prerequisites and interdependencies. Understanding which courses to take and in what order can be overwhelming, particularly for those early in their studies or unfamiliar with the curriculum structure.

The project addresses these issues by providing Degree Companion, a web-based degree progress manager designed to support students throughout their academic journey. The core motivation behind this initiative centers on three key objectives. Firstly, the tool aims to increase student motivation and engagement by offering a clear overview of progress and upcoming milestones, helping students remain focused on their goals. Secondly, it seeks to ensure a structured and coherent learning path by helping students understand course dependencies and plan a logical sequence of modules. Thirdly, it strives to reduce common pitfalls and stressful periods through proactive planning support that minimizes the risk of missed prerequisites and last-minute scheduling conflicts. By tackling these challenges, Degree Companion aims to transform the often-stressful experience of degree planning into an organized and empowering process.

Degree Companion is written in TypeScript and uses Tauri to create executable packages for Windows, Mac, and Linux. This project report focuses on the implementation and deployment of Degree Companion.

Chapter 2

Setup and Build Procedure

Degree Companion is an open-source application built with modern web technology such as TypeScript, Svelte, and Tauri. The source code is available on GitHub [Burtscher et al. 2026]. The application itself is available as a binary package for Windows, Mac, or Linux. Due to the use of Tauri-specific plugins (for example, File System, Dialog), it is not available as a regular web application.

The application is written in TypeScript [Microsoft 2026] and uses Svelte [Svelte 2026] for interfaces and core components. Tauri v2 [Tauri 2026b] is a framework which enables the use of modern web technology to build standalone applications for three common desktop platforms: Windows, MacOS, and Linux. In contrast to frameworks like Electron [OpenJS 2026], Tauri supports interacting with system APIs through a Rust interface [Tauri 2025c], allowing more seamless integrations and faster operations by utilizing the system hardware in a more direct manner. A functional Rust toolchain is required to use Tauri.

Bun was chosen as the package manager for the project [Bun 2026a], due to its low overhead and fast operations. Nonetheless, since it uses similar syntax for commands, it may also be possible to interchange Bun with other npm-like package managers. Finally, as with most modern open-source software projects, Git [Git 2026] is used for software management and version control.

The command `bun run tauri dev` can be used to try out the application temporarily. When executing this command for the first time, it may take a while to download and compile any packages/crates required. The command `bun run tauri build` creates a standalone executable file for the currently used operating system in the folder `src-tauri/target/`.

To streamline and simplify application development workflow, the repository also contains a GitHub Actions workflow configuration to automatically build executable packages for the three common desktop platforms. The workflow is based on the example template published in the Tauri documentation [Tauri 2025a; Tauri 2026a]. Some tweaks were necessary for the Degree Companion tech stack. These changes included changing any usage of Node/npm to use corresponding Bun steps instead [Bun 2026b]. Additionally, automatic release compilation for ARM-based Linux distributions was removed due to failing to install all necessary dependencies. Furthermore, there seem to be issues with the speed of compilation for Tauri projects on ARM-based Linux, which drastically increase the computational time required by the GitHub Runners [Tauri 2025a]. Lastly, for convenience, the configuration was changed to always run a pipeline when pushing to main, instead of a separate release branch. The most important changes can be seen in Listing 2.1.

Finally, it is also important to note that it is not possible at the moment to deploy the Degree Companion application as a regular web app, due to the use of Tauri-specific plugins (for example, File System, Dialog). The core application logic would need to be restructured to build a version of the application deployable on web servers, and issues issues around persistency of user data would have to be solved.

```yaml
...
on:
  workflow_dispatch:
  push:
    branches:
      - main
...
# Switched from Node to bun setup step
- name: Install bun
        uses: oven-sh/setup-bun@v2
        with:
          bun-version: latest
...
# Switched from npm to bun install step
- name: install frontend dependencies
        run: bun install
```

Listing 2.1: Modifications made to Tauri workflow.

Chapter 3

Implementation

The Degree Companion application combines modern web technologies with native desktop capabilities to deliver a seamless user experience.

3.1 Technology Stack

The foundation of Degree Companion is a modern technology stack chosen for performance, reactivity, and a native desktop experience.

3.1.1 Frontend Framework

The frontend is built using SvelteKit, a framework for developing web applications using Svelte [Svelte 2025a]. Svelte serves as the reactive UI framework. It uses a compiler to turn declarative components written in HTML, CSS, and JavaScript into optimized code [Svelte 2024]. The compilation step happens during build time, resulting in JavaScript that updates the DOM efficiently.

The application uses Svelte's component-based architecture to create reusable UI elements such as CourseCard, SemesterNode, and CurriculumTree. Svelte 5 introduces a runes API that provides explicit reactivity through compiler instructions [Svelte 2025b]. Runes inform Svelte about reactivity, syntactically represented as functions starting with a dollar sign. This approach to reactivity enables the application to handle complex state updates across multiple components.

3.1.2 Desktop Integration

Tauri provides a native desktop wrapper, enabling a web application to run as a standalone desktop application with access to system-level APIs [Tauri 2025c]. Tauri is a framework for building binaries for major desktop platforms that allows developers to integrate frontend frameworks that compile to HTML, JavaScript, and CSS. Tauri uses the operating system's native webview rather than bundling a browser engine, which results in smaller application sizes and lower memory usage compared to alternatives like Electron [OpenJS 2026]. The choice of Tauri was also driven by its security-first design, which isolates the webview process to reduce the application's potential attack surface [Tauri 2025b]. This architecture ensures that the application is not only lightweight but also robust and secure.

A number of Tauri plugins are integrated into the Degree Companion application:

- `@tauri-apps/plugin-dialog` for native file dialogs.
- `@tauri-apps/plugin-fs` for filesystem operations.
- `@tauri-apps/plugin-opener` for opening external links.
- `svelte-tauri-filedrop` for drag-and-drop file handling.

The use of these plugins mean that it is not possible to deploy the Degree Companion application as a regular web app, but only as a desktop executable package.

3.1.3 Visualization and Data Processing

The curriculum visualization component of Degree Companion uses Svelte Flow (also known as `@xyflow/svelte`) for rendering interactive curriculum graphs [webkid 2026; Gorzo and Klack 2025]. This library provides a framework for creating node-link-based interfaces such as workflow builders and data visualizers [Robb 2023]. The library allows users to visualize course dependencies and semester planning through an interactive node-link-based interface.

PapaParse handles CSV parsing for data import functionality [Holt 2026a; Holt 2026b]. PapaParse is a CSV parser for the browser that supports web workers and streaming of large files [Holt 2025]. Degree Companion uses a custom CSV format to represent curriculum information, which was developed specifically for this application and is not based on any existing standard. The custom data format defines the following fields for each course entry:

- `course_id`: Unique identifier for the course (e.g., 710.000, 705.001).
- `course_name`: Full name of the course.
- `module_code`: Code identifying the module to which the course belongs (e.g., A, B, C).
- `module_name`: Full name of the module (e.g., "Fundamentals of Computer Science").
- `course_subcategory`: Optional subcategory within the module (e.g., "Algorithms").
- `course_type`: Type of course format (e.g., VO for lecture, KU for exercise).
- `credits`: Number of ECTS credits awarded for the course.
- `required_code`: Indicates whether the course is required or elective depending on degree.
- `availability`: Semester availability (W for winter semester, S for summer semester).
- `recommended_semester`: The semester in which the course is typically taken.
- `prerequisites`: Course IDs of prerequisite courses, separated by semicolons.
- `frequency`: How often the course is offered (e.g., yearly, semester).
- `language`: Language of instruction (DE for German, EN for English).
- `description`: Brief description of the course content.
- `url`: Link to the official course page in the university's course management system.

This structured format enables the application to parse curriculum data systematically and establish relationships between courses, modules, and prerequisites. The format captures nearly all essential information needed for curriculum planning, including temporal constraints (availability and recommended semester), academic requirements (prerequisites and credits), and organizational structure (modules and subcategories). Users can import their own curriculum data by creating CSV files that conform to this format specification.

3.2 Application Architecture

The application architecture follows a structure that emphasizes separation of concerns and maintainability. The architecture is built around a reactive state management system that ensures data consistency

```
export const curriculumStore = writable<Curriculum>({
  credits: 0,
  modules: [],
  courses: [],
  degreeType: 'bachelor',
  startSemester: 'winter',
  majorModule: '',
  minorModule: ''
});

export const graphStore = writable<Graph>({
  nodes: [],
  edges: [],
  strokeWidth: 2,
  strokeColor: '#000000',
  semesterCount: 0,
  courseCardStates: {}
});
```

Listing 3.1: Two stores are used to manage application state.

across all components, complemented by a persistence layer that preserves user work across sessions. This section explores the key architectural components that enable the application's functionality.

3.2.1 State Management

The application employs Svelte's built-in store system for centralized state management. This approach provides a single source of truth for application data, while maintaining Svelte's reactivity model. Two primary stores manage the application state, the `curriculumStore` and the `graphStore`, as shown in Listing 3.1. The `curriculumStore` maintains curriculum-level data, including course lists, module information, and degree configuration. The `graphStore` manages the visual representation of the curriculum as a node-edge graph. This separation allows the application to maintain both the logical structure of the curriculum and its visual representation independently. The store architecture follows Svelte's reactive principles, where components can subscribe to store changes and automatically re-render when relevant data updates. This eliminates the need for manual change detection and reduces the likelihood of stale data being displayed to users.

3.2.2 State Persistence and Synchronization

The application includes a storage manager that handles application state persistence using browser localStorage. The system implements automatic saving with debouncing to prevent excessive write operations, as shown in Listing 3.2. The storage manager handles complex data structures through custom serialization functions that manage circular references and ensure data integrity. The `initAutoSave()` method subscribes to store changes and triggers debounced saves after a 1000 ms delay. This debouncing mechanism ensures that rapid successive changes (such as dragging a course node across the graph) do not trigger excessive localStorage writes.

The serialization process addresses the fact that the graph structure contains circular references between nodes and edges. Custom serialization functions extract only the necessary data for persistence, converting object graphs into JSON-serializable structures, while maintaining referential integrity through identifier-based lookups. The auto-save functionality ensures that changes are persisted to localStorage without requiring explicit save actions. This allows users to close and reopen the application without losing their work. Synchronization operates through Svelte's store subscriptions, which notify the storage manager

```
export const storageManager = {
  save() {
    if (!autoSaveEnabled) return;
    try {
      const curriculum = get(curriculumStore);
      const graph = get(graphStore);

      localStorage.setItem(STORAGE_KEY, JSON.stringify({
        curriculum: {
          ...curriculum,
          courses: curriculum.courses.map(serializeCourse)
        },
        graph: {
          ...graph,
          nodes: graph.nodes.map(serializeNode),
          edges: graph.edges.map(serializeEdge)
        },
        csv: get(csv),
        timestamp: Date.now()
      }));
    } catch (error) {
      console.error('Failed to save state:', error);
    }
  }
}
```

Listing 3.2: The storage manager's save implementation.

when state changes occur. The debounced save operation ensures consistent performance during intensive interaction sessions.

3.2.3 State Recovery and Data Loading

The application implements a state recovery mechanism that deserializes previously saved data and reconstructs object relationships. The implementation is shown in Listing 3.3. This approach rebuilds necessary lookup maps (`courseMap` and `moduleMap`) to restore object references that cannot be directly serialized to JSON. The deserialization process ensures that all relationships between courses, modules, and graph elements are properly reconstructed. The function returns a boolean indicating success or failure, allowing the application to handle corrupted or incompatible saved states.

3.2.4 State Reset

The application provides a mechanism to reset the application state with the function `clear()`, shown in Listing 3.4. This function temporarily disables auto-save, clears localStorage, resets all stores to their initial values, and re-enables auto-save after a brief delay. The `resetKey` update triggers re-initialization of components that depend on the reset state, allowing the user to restart the workflow from the data import stage.

3.3 Application Workflow

A typical user journey goes from initial application launch, through data import, configuration, and visualization of the curriculum graph. The workflow follows a three-stage process that guides users from raw data to a fully interactive curriculum plan.

```
load(): boolean {
  try {
    const saved = localStorage.getItem(STORAGE_KEY);
    if (!saved) return false;

    const state = JSON.parse(saved);
    const courses = deserializeCourses(
      state.curriculum.courses,
      state.curriculum.modules
    );
    const courseMap = buildMap(courses, c => c.id);
    const moduleMap = buildMap(state.curriculum.modules, m => m.code);

    curriculumStore.set({ ...state.curriculum, courses });

    if (state.graph) {
      graphStore.set({
        ...state.graph,
        nodes: state.graph.nodes.map(
          (n: any) => deserializeNode(n, courseMap, moduleMap)
        ),
        edges: state.graph.edges.map(
          (e: any) => deserializeEdge(e, courseMap)
        )
      });
    }

    csv.set(state.csv);
    return true;
  } catch (error) {
    console.error('Failed to load state:', error);
    return false;
  }
}
```

Listing 3.3: State recovery mechanism.

3.3.1 Initial Loading and State Detection

When the application launches, the main page component checks for an existing saved state to determine whether to present the initial data import interface or restore a previous session. The initialization logic is shown in Listing 3.5. This workflow ensures that returning users are immediately directed to their curriculum graph if valid data exists, while new users are presented with the data import interface. The `isOnboardingComplete()` function validates that the curriculum store contains the minimum required data for a functional curriculum plan. If the stored state is incomplete or corrupted, the application clears it and presents a fresh start.

3.3.2 Data Import Process

The first step in creating a new curriculum plan involves importing course data into the system. The application provides a Dropzone component on the root page, that accepts the custom CSV files containing course information. Users can either drag and drop curriculum data files or use a file picker dialog to select files from their system. The files are then parsed, effectively validating the CSV structure and ensuring that required fields are present with correct data types.

The CourseListTable component displays all imported courses in a tabular format, showing metadata

```
clear() {
  try {
    autoSaveEnabled = false;
    localStorage.removeItem(STORAGE_KEY);

    curriculumStore.set({
      credits: 0,
      modules: [],
      courses: [],
      degreeType: 'bachelor',
      startSemester: 'winter',
      majorModule: '',
      minorModule: ''
    });

    graphStore.set({
      nodes: [],
      edges: [],
      strokeWidth: 2,
      strokeColor: '#000000',
      semesterCount: 0,
      courseCardStates: {}
    });

    csv.set(undefined);
    resetKey.update(n => n + 1);
    setTimeout(() => autoSaveEnabled = true, 100);
  } catch (error) {
    console.error('Failed to clear state:', error);
    autoSaveEnabled = true;
  }
}
```

Listing 3.4: The `clear()` function resets application state.

```
onMount(async () => {
  if (storageManager.hasState()) {
    storageManager.load();

    if (isOnboardingComplete(get(curriculumStore))) {
      await goto('/graph');
    } else {
      storageManager.clear();
    }
  }
  isCheckingStorage = false;
});
```

Listing 3.5: Checking for saved state upon application initialization.

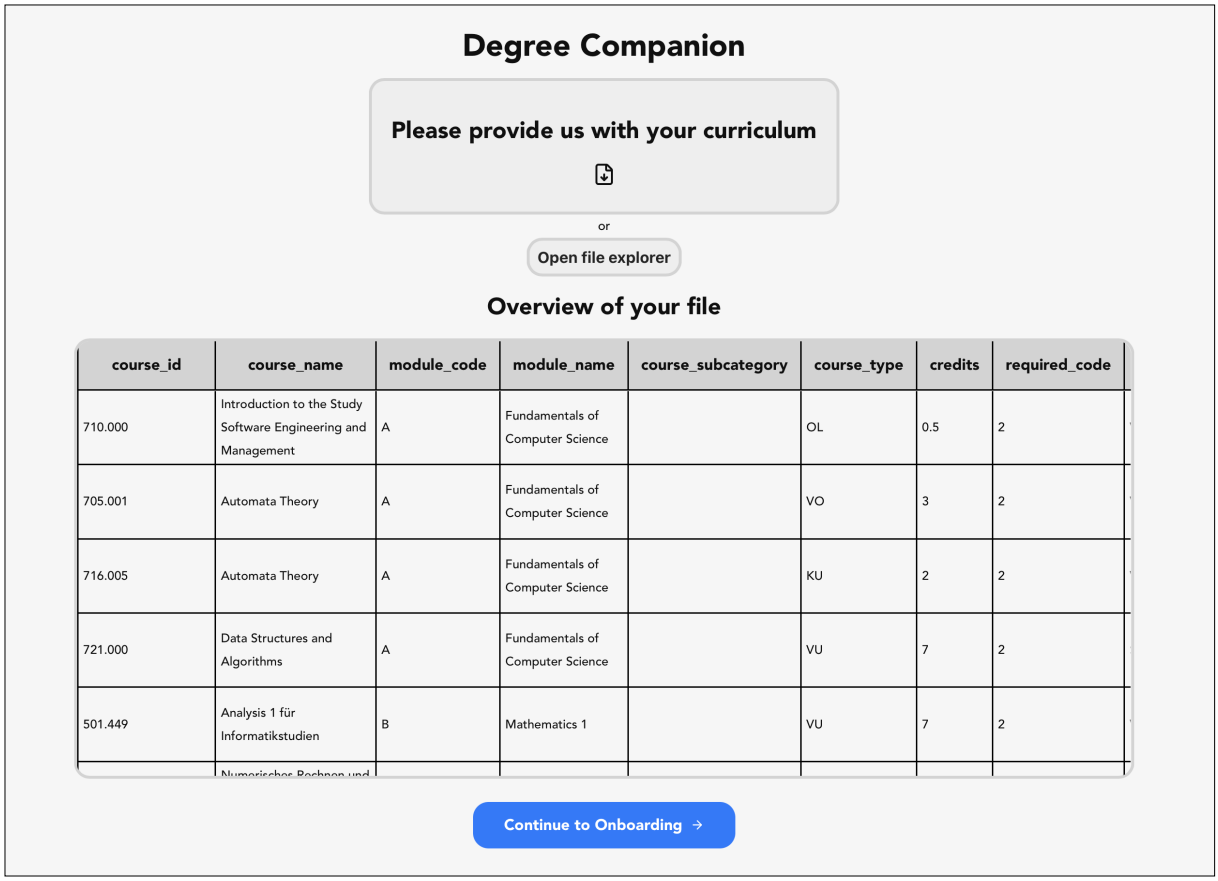

Figure 3.1: Degree Companion: Data import interface showing dropzone and course list table preview.

such as course codes, titles, credits, and module assignments. This preview allows users to verify the accuracy of the imported data. As shown in Figure 3.1, the interface presents both the dropzone area for file selection and the resulting course list table after a CSV file has been processed. The table provides a scrollable interface for locating specific courses and checking for missing information. This review step allows users to identify potential import errors before committing the data. Once confirmed, users proceed to the onboarding configuration stage.

3.3.3 Onboarding and Configuration

After the successful import and verification of the course data, the user proceeds to the onboarding configuration stage. This step is critical, as it transforms the flat list of courses into a structured, personalized academic plan. The Onboarding form, shown in Figure 3.2, is presented to users configure the essential parameters that define their curriculum. The Onboarding form collects the degree type (Bachelor's or Master's), the intended start semester (winter or summer), and the total ECTS credits required for the degree. Depending on the degree type, it then prompts for the ECTS breakdown per module.

The form incorporates real-time validation to ensure data integrity and prevent the creation of an invalid curriculum. The button to proceed to curriculum graph visualization remains disabled until all conditions are met. The validation logic differs based on the selected degree type:

- *Bachelor's Degree*: The system requires the user to specify the ECTS for each individual module listed and extracted from the CSV. It mandates that a positive ECTS value is assigned to every module.

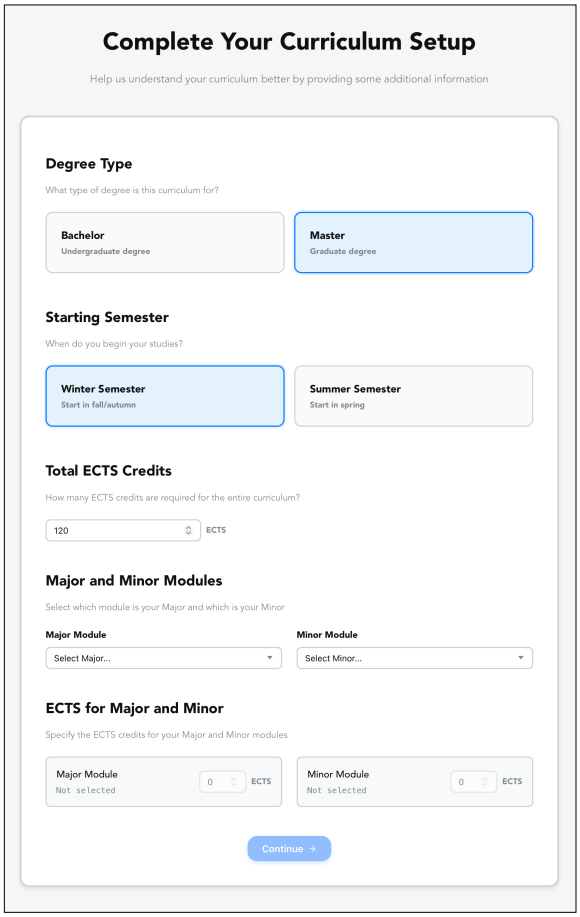

Figure 3.2: Degree Companion: Onboarding interface for curriculum configuration.

- *Master's Degree*: The system requires that the user selects two *distinct* modules for their major and minor fields of study. Additionally, it requires a positive ECTS value to be specified for both the selected major and minor modules.

This proactive validation guides the user toward a valid configuration before they can proceed. Once all inputs are validated, the configuration data is committed to the `curriculumStore`, where it directly influences how the curriculum graph is generated and displayed.

3.3.4 Graph Visualization and Interaction

The curriculum graph provides users with a visual representation of their academic plan. The graph is rendered using custom Svelte components that extend the `@xyflow/svelte` library. The application defines several specialized node types to represent different elements of the curriculum:

- `SemesterNode`: Represents a semester container that can hold multiple courses, providing temporal organization to the curriculum plan.
- `CustomNode`: Represents individual course cards with metadata including course code, title, credits, and completion status.
- `HorizontalSeparatorNode` and `VerticalSeparatorNode`: Provide visual organization and grouping within the graph layout.
- `AddSemesterNode`: Allows users to dynamically add new semesters to their plan.

Custom edges (`CustomEdge.svelte`) connect courses to visualize prerequisites and course sequences.

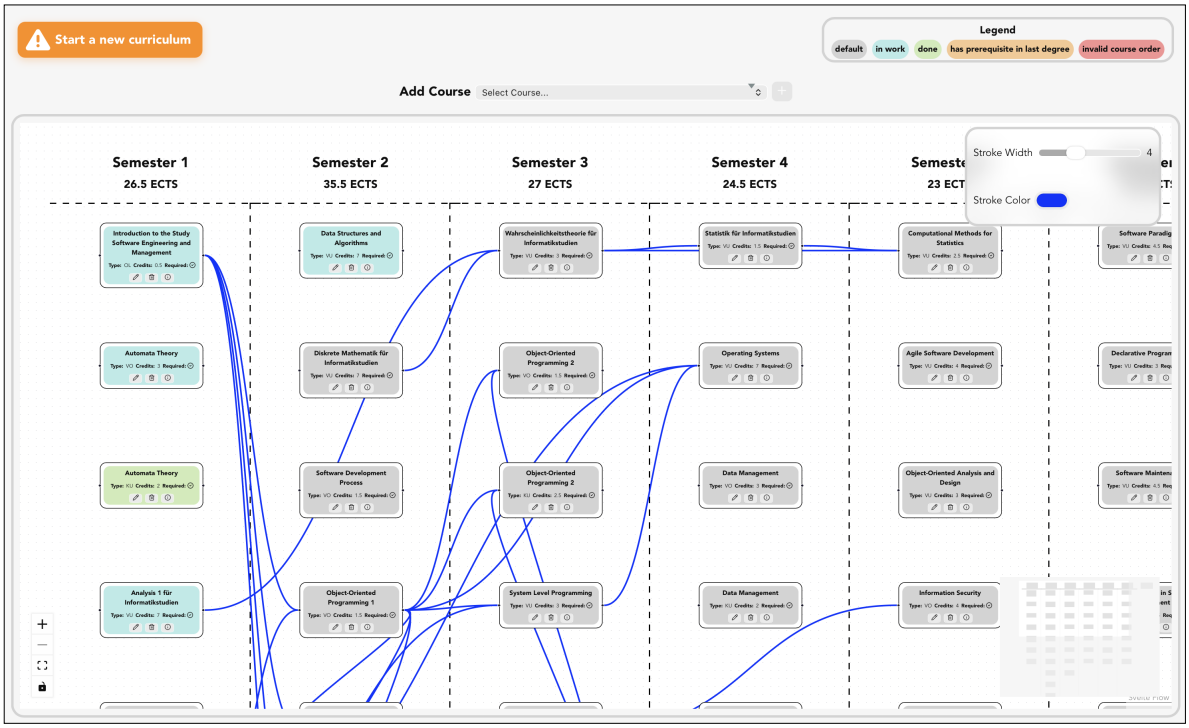

Figure 3.3: Degree Companion: An interactive curriculum graph with course nodes and prerequisite edges.

The graph supports interactive features such as drag-and-drop course placement, dynamic edge creation, and real-time updates to the underlying data model. Users can reorganize their curriculum by dragging courses between semesters.

The graph visualization interface can be seen in Figure 3.3, showing a curriculum graph with multiple semesters, course nodes, and prerequisite connections. The visualization interface provides controls for manipulating the graph layout and adjusting course placements. The visualization system maintains bidirectional synchronization between the graph representation and the underlying data stores, ensuring that changes made through the visual interface are immediately reflected in the curriculum data and vice versa. This synchronization is achieved through Svelte's reactive store system, as described in Listing 3.1.

3.4 Component Architecture

The application follows a modular component structure with clear separation of concerns. Each component has a defined responsibility and communicates through the Svelte stores for shared application state.

3.4.1 Component Organization

The component hierarchy is organized into four primary categories:

- *Layout Components*: Handle application-wide structure and navigation, providing header, footer, and routing functionality.
- *Data Components*: `Dropzone`, `CourseListTable`, and `CourseList` manage data import and display, encapsulating CSV parsing and validation logic.
- *Graph Components*: `CurriculumTree`, various node types, and `CustomEdge` handle visualization, implementing the interactive graph interface.

- *UI Components*: CourseCard and Legend provide reusable interface elements that maintain visual consistency.

3.4.2 Custom Node Implementation

The graph visualization relies on custom node components that extend `@xyflow/svelte`. The SemesterNode component acts as a container managing course node layout within semester boundaries, handling drag-and-drop events for course movement between semesters.

The CustomNode component represents individual courses, displaying course code, title, credit value, and completion status. The node's visual appearance adapts based on course state, using different colors to indicate completion status.

Separator nodes (HorizontalSeparatorNode and VerticalSeparatorNode) provide visual organization, while AddSemesterNode allows users to dynamically add new semester containers.

3.4.3 Edge Rendering and Prerequisites

The CustomEdge component visualizes prerequisite relationships between courses. Users can click on an edge to reveal the origin and target node, making the prerequisite flow clear. The system handles complex relationships, including courses with multiple prerequisites or serving as prerequisites for multiple subsequent courses.

3.4.4 Reactive Data Flow

The component architecture leverages Svelte's reactive data flow to maintain consistency between visual representation and underlying data. User interactions trigger updates through the reactive system, with the graph component updating `graphStore`, which triggers `curriculumStore` updates. This ensures the visual state and the data state remain synchronized.

The reactive system handles derived state calculations, such as computing semester credits or validating prerequisites, automatically, whenever relevant state changes occur.

Chapter 4

Future Work

The current Degree Companion application serves as a functional prototype for degree planning. Several technical refinements and feature expansions remain to be explored. The project's evolution should focus on transitioning from manual configuration toward a more automated, intelligent system capable of supporting diverse academic paths across multiple studies and institutions.

To optimize the user experience, future adaptations should automate the integration of onboarding data into the core graph logic. Reducing the manual overhead of initial module configuration setup can be achieved by storing credits per module in a separate, more advanced data format (for example, a zip file containing multiple CSV files). An integrated table editor could be implemented to modify parsed data directly within the application.

Beyond data integration, the user interface could improve user guidance throughout the application. Potential future features include filter options when adding new courses, allowing users to refine the course search by detailed information (e.g., lecture type, credit count). This should help users select courses that are actually relevant and suitable for their study. An integration of a recommendation for new lectures based on already finished courses could help reduce the search for future courses.

Future development should focus on expanding the application's integration with existing academic services. To maximize accessibility, the application must evolve beyond a desktop based on Tauri. Seamless integration into the university's campus management system, TUGRAZonline, specifically through a potential cooperation with CampusOnline, could help embed the service with existing applications. The core application logic would therefore need to be restructured to build a version of this application deployable on web servers.

To increase the platform's versatility and reusability, future development should enable the management of multiple concurrent curriculum plans. This functionality is essential for students enrolled in more than one degree or those transitioning from a Bachelor's to a Master's program. Furthermore, the utility of the tool can be extended by exporting the degree graph in standardized formats such as SVG, PNG, or PDF.

Gathering feedback from users is essential to evaluate and improve the system. While the current approach only supports the two mentioned degrees from TU Graz, it should be evaluated whether curriculum data from other universities can be summarized in the proposed data format.

Chapter 5

Concluding Remarks

The development of Degree Companion has resulted in a comprehensive prototype that demonstrates how an interactive degree planning tool can help simplify complex academic planning problems. Intense conceptual planning was required to find a suitable, universal curriculum data format. Significant manual effort was invested into modeling data from one Bachelor's and one Master's degree in this system. The resulting custom CSV format successfully captures essential metadata such as credit counts, course types, and complex prerequisite structures.

Degree Companion implements an interactive node-based visualization approach that makes course interdependencies transparent to the user. By using SvelteKit and Tauri, the project is available for multiple platforms as a desktop application. Students can manage course selection, while visualizing semester progress and keeping track of academic overhead. Ultimately, Degree Companion has demonstrated the potential for a tool to help reduce the stress of degree progression planning. While the current prototype is tailored to two specific degrees at Graz University of Technology, the application could be evaluated to possibly extend the usability for other degrees and institutions.

Bibliography

Bun [2026a]. *Bun - A Fast All-In-One JavaScript runtime*. Anthropic, 01 Feb 2026. https://bun.com/ (cited on page 3).

Bun [2026b]. *Install and Run Bun in GitHub Actions*. Anthropic, 01 Feb 2026. https://bun.com/docs/guides/runtime/cicd (cited on page 3).

Burtscher, Christian, Jonas Glaser, Marcus Gugacs, and Eva Haring [2026]. *Degree Companion*. Version 0.3.0. 30 Jan 2026. https://github.com/gugacs/degree-companion (cited on page 3).

Git [2026]. *Git*. Software Freedom Conservancy, 01 Feb 2026. https://git-scm.com/ (cited on page 3).

Gorzo, Peter and Moritz Klack [2025]. *Svelte Flow 1.0 Is Here!* May 2025. https://xyflow.com/blog/svelte-flow-release (cited on page 6).

Holt, Matt [2025]. *Parse CSV with JavaScript*. May 2025. https://npmjs.com/package/papaparse (cited on page 6).

Holt, Matt [2026a]. *Papa Parse Documentation*. 30 Jan 2026. https://papaparse.com/docs (cited on page 6).

Holt, Matt [2026b]. *Papa Parse: The Powerful In-Browser CSV Parser*. 30 Jan 2026. https://papaparse.com (cited on page 6).

Microsoft [2026]. *TypeScript: JavaScript with Syntax for Types*. Microsoft, 01 Feb 2026. https://typescriptlang.org/ (cited on page 3).

OpenJS [2026]. *Build Cross-Platform Desktop Apps with JavaScript, HTML, and CSS*. OpenJS Foundation, 01 Feb 2026. https://electronjs.org/ (cited on pages 3, 5).

Robb, John [2023]. *Svelte Flow - A Library For Rendering Interactive Node-Based UIs*. Nov 2023. https://xyflow.com/blog/svelte-flow-launch (cited on page 6).

Svelte [2024]. *Svelte Documentation Overview*. Oct 2024. https://svelte.dev/docs/svelte (cited on page 5).

Svelte [2025a]. *Introduction to SvelteKit*. Mar 2025. https://svelte.dev/docs/kit/introduction (cited on page 5).

Svelte [2025b]. *Svelte 5 Migration Guide*. Nov 2025. https://svelte.dev/docs/svelte/v5-migration-guide (cited on page 5).

Svelte [2026]. *Svelte: Cybernetically Enhanced Web Apps*. 30 Jan 2026. https://svelte.dev (cited on page 3).

Tauri [2025a]. *GitHub | Tauri*. Aug 2025. https://v2.tauri.app/distribute/pipelines/github/ (cited on page 3).

Tauri [2025b]. *Tauri's Security Model*. Feb 2025. https://v2.tauri.app/security/ (cited on page 5).

Tauri [2025c]. *What Is Tauri?* Sep 2025. https://v2.tauri.app/start/ (cited on pages 3, 5).

Tauri [2026a]. *Tauri GitHub Action*. 01 Feb 2026. https://github.com/tauri-apps/tauri-action (cited on page 3).

Tauri [2026b]. *Tauri: Create Small, Fast, Secure, Cross-Platform Applications*. 30 Jan 2026. https://v2.tauri.app/ (cited on page 3).

webkid [2026]. *xyflow: Node-Based UIs for React and Svelte*. 30 Jan 2026. https://xyflow.com/ (cited on page 6).