Leveraging Large Language Models to Support Functional Programming Education

A Learning Assistant for Haskell

by

Ahmad Khdeir

August 12, 2025

A document submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

at

Goethe University Frankfurt

Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich die vorliegende Bachelorarbeit selbstständig und ohne unerlaubte Hilfe angefertigt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Die Arbeit wurde in gleicher oder ähnlicher Form noch nicht in einem anderen Prüfungsverfahren vorgelegt.

Ort, Datum: Frankfurt am Main, 13.08.2025

Unterschrift: 

Acknowledgments

I would like to express my sincere gratitude to my supervisor Alperen Kantarcı for his guidance and support throughout this thesis. I also thank Prof. Dr. Visvanathan Ramesh for his insightful feedback. Special thanks to my friends and family for their continuous encouragement. This project would not have been possible without their help and motivation.

Abstract

Learning functional programming in Haskell poses unique challenges—abstract reasoning, symbolic manipulation, and deciphering unfamiliar error messages—that often overwhelm novice students. This thesis explores whether a purpose-built, AI-supported environment can mitigate these hurdles. We present **Haskify**, a web application that combines a syntax-aware Haskell editor, a real-time GHC-based compiler, PDF exercise upload, and a domain-specialized "scaffold-not-solve" AI assistant that explains concepts and nudges learners without revealing full solutions. We evaluated Haskify in a single-condition pilot usability study with 23 students from the Programming Paradigms and Compiler Construction (PPDC) course. Participants completed short Haskell exercises in the app and then submitted a brief questionnaire. We measured task success, perceived understanding, ease of use (1–4), and reuse intention. Results indicate that 73.9% fully solved the assigned tasks and 21.7% solved them partially; 87.0% reported that Haskify helped them understand the problems better. Ease-of-use ratings clustered at the top of the scale (most ratings were 3/4 or 4/4), and 91.3% indicated they would use Haskify again for learning or exam preparation. These findings suggest that integrating live compiler feedback with a constructivist, domain-focused AI tutor can meaningfully support novice learners in functional programming. We discuss limitations of this pilot (small sample, single-condition design, and free-tier latency) and outline future work, including a controlled comparison against a static baseline, the addition of formal cognitive-load measures, and enhancements to the hint taxonomy and instructor tooling.

Contents

1 Introduction

1.1 Why Functional Programming?

Many universities opt for functional programming as the introductory course for first-year computer science students due to several compelling reasons. Functional programming provides an excellent foundation in algorithms, as it allows algorithms to be expressed clearly and concisely, avoiding unnecessary syntax that can often complicate understanding and implementation [5]. Another significant advantage of introducing functional programming early in the curriculum is its emphasis on algorithmic design rather than syntactic details. While imperative programming continues to dominate the field of computer science, teaching alternative programming paradigms such as functional programming is beneficial to broaden students' perspectives. Additionally, many professors and institutions favor functional programming for beginners because it naturally introduces students to essential areas like algebra, artificial intelligence, formal language theory, and specification, making it an effective starting point for these critical topics. Furthermore, it's important to acknowledge that Haskell is one example of a functional programming language.

Why this matters for students. For first-year learners, the payoff of starting with a functional mindset is practical: it cultivates mathematical thinking, immutability by default, and a habit of reasoning about programs as pure functions. These habits travel well—into testing, parallelism, and safe refactoring later in the degree—and reduce "accidental complexity" that often derails beginners in their first months of coding.

1.2 Challenges with Learning Functional Programming

Now, let's discuss some common problems students face while learning programming in general. Many students struggle with understanding abstract concepts, such as how loops function or how methods execute during program runtime [3]. Another significant issue is students' difficulty comprehending programming structures and the process of building a complete program. Additionally, mastering the syntax of programming languages is another frequent challenge. Merely understanding pseudo-code is insufficient; students must also

grasp specific syntax rules, which can become especially confusing when learning multiple languages simultaneously. Hence, beyond conceptual understanding, practical application and experimentation with code are critical. Another difficulty students encounter is the lack of sufficient examples provided during lectures. Although this may be intentional, as instructors often encourage independent research, it can be particularly confusing for first-year students who might require more structured guidance. Additionally, available examples sometimes fail to address all possible cases discussed in class. Furthermore, students frequently face challenges with complex data structures [8]. This complexity often prompts numerous questions that require immediate clarification. However, tutors or professors are rarely available at all times, and providing instantaneous, real-time answers is practically challenging.

So what is missing for learners? In practice, students need three things at once: (1) a place to try ideas safely and see concrete compiler feedback, (2) targeted explanations that translate abstract concepts into the exact code they just wrote, and (3) help that arrives *when the confusion happens*, not hours later on a forum or in office hours.

1.3 Proposed Solutions

Here comes Haskify—but what exactly is Haskify? How does it help, and what does it provide? Haskify is a web application featuring a Haskell code editor with syntax highlighting and an integrated real-time compiler. More importantly, it includes an AI assistant specifically trained for functional programming. Unlike typical general-purpose AI tools such as ChatGPT, Haskify's AI is highly specialized in Haskell and functional programming, providing real-time, around-the-clock support. This AI assistant has been carefully trained to focus exclusively on functional programming topics, particularly Haskell, ensuring that unrelated queries are not addressed. Even for relevant queries, the AI emphasizes detailed explanations and guidance rather than providing direct solutions immediately. This approach mirrors the role of an experienced university tutor, promoting deeper learning and understanding. Therefore, Haskify serves as an essential tool for anyone aiming to learn Haskell and functional programming effectively. Users can write code directly in the editor, where the AI assistant promptly identifies and explains errors—be they syntax, logical, or conceptual. Crucially, the assistant guides users through their mistakes by offering insightful hints and constructive explanations without simply handing over the answers, preserving the joy and fulfillment inherent in the learning process.

Why students need Haskify

- **Immediate, contextual help:** Feedback arrives at the exact line and moment of confusion, closing the gap between lecture theory and real code.
- **From error to understanding:** Compiler messages are unpacked into plain language and tied to the student's snippet, turning cryptic types or pattern-match errors into teachable moments.
- **Scaffold—not—solve:** The assistant nudges with hints and micro-explanations instead of dropping full solutions, helping learners build independence (and protecting academic integrity).
- **Low-friction practice:** No heavy IDE setup—open the browser, type Haskell, compile, iterate.
- **Confidence and momentum:** Rapid feedback loops reduce frustration and help students stay engaged long enough to form correct mental models.

What's special about Haskify

- **Domain-specialized AI:** Tuned specifically for Haskell and functional concepts, avoiding off-topic tangents common with general chatbots.
- **Live compiler at the core:** Explanations are grounded in real runs via GHC, not just guesses.
- **Editor intelligence:** Syntax highlighting and guardrails via Monaco + a Haskell tokenizer provide immediate visual cues and reduce syntactic thrash.
- **Deployment you can rely on:** A Dockerized backend on Render and a Vercel-built frontend make it easy for students and instructors to access the same, consistent environment.

A day-in-the-life example. A student implements a recursive function, gets a pattern-match warning, and sees a confusing type error. Haskify highlights the line, explains what the compiler means, asks a guiding question about the missing base case, and offers a tiny example—not the entire solution—so the student corrects it and re-runs immediately. The concept sticks because the fix is *their* code.

1.4 Research Question

This thesis is guided by the following central research question:

> *How can an AI-assisted educational platform specifically designed for Haskell support students in overcoming common challenges when learning functional programming?*

The objective is to investigate whether integrating a specialized AI assistant—focused on conceptual guidance rather than direct problem-solving—can meaningfully improve learners' engagement, comprehension, and independence in tackling functional programming tasks. The development and evaluation of the *Haskify* system aim to provide both a technical implementation and a pedagogical reflection on this approach.
Operationalization (at a glance). To answer this question, the thesis (i) implements the end-to-end system, (ii) articulates design principles that embody scaffolded support, and (iii) sets up an evaluation plan comparing Haskify-guided work with a static tutorial baseline on effectiveness, efficiency, and perceived understanding.

1.5 Thesis Structure

In this thesis, we present the development of Haskify, a web-based educational platform designed to support students in learning functional programming through Haskell. By combining an integrated code editor, real-time compiler, and an AI assistant trained for domain-specific support, Haskify aims to make functional programming more accessible and engaging. The remainder of this thesis is structured as follows: Chapter 2 reviews related work. Chapter 3 states system requirements and design. Chapter 4 describes the implementation. Chapter 5 presents the evaluation design (and results). Chapter 6 offers a didactic reflection. The thesis closes with an Artifacts & Code Availability note and the bibliography.

2 Related Work

2.1 AI-Assisted Tools in Computer Science Education

2.1.1 CS50 Duck Debugger

Before diving into the technical implementation and technology stack behind Haskify, it is essential to examine related tools that served as sources of inspiration. Some of these tools sparked innovative ideas that shaped Haskify's design, while others highlighted limitations that this project aims to address or improve upon. One particularly noteworthy example is the *CS50 Duck Debugger (DDB)*, a tool developed in the summer of 2023 at Harvard University specifically for students enrolled in the CS50 course. A key motivation behind the creation of this tool was to simulate a 1:1 teacher-to-student ratio—an educational challenge also discussed in the introduction of this thesis [7]. The DDB offers three core functionalities: (1) it provides explanations for highlighted code snippets, (2) it evaluates code style with enhanced feedback, and (3) it includes a chatbot capable of answering questions related to the CS50 curriculum. A particularly important aspect of the system is its use of a dedicated API endpoint that controls the output of GPT-4, ensuring that all AI-generated responses align closely with the goals and content of the CS50 course. This integration is made possible through a *partnership with OpenAI*, where the OpenAI API powers the backend AI functionality. One feature that stood out during analysis is the implementation of a guard mechanism against prompt injection attacks. This preventive measure is designed to stop users from manipulating the AI into delivering direct solutions to homework assignments—an issue of academic integrity that is especially relevant in educational contexts.

2.1.2 GitHub Copilot

Another significant tool worth highlighting is GitHub Copilot, a powerful AI-assisted coding tool that caters to both novice and experienced programmers. As noted by Puryear [9], Copilot can generate code with a human-graded score ranging between 68% and 95%, which is quite remarkable. However, it is important to acknowledge that GitHub Copilot is also powered by OpenAI's code-focused language models. While Copilot was not designed specif-

ically as an AI education tool, it is widely adopted by learners to practice programming and sharpen their skills, making it a valuable point of discussion in the context of educational technology. Due to its ease of use, Copilot has the potential to fundamentally reshape how programming is taught and learned. In fact, it has been speculated that Copilot may be capable of completing CS1-level assignments with greater accuracy than the students enrolled in such introductory courses [9]. This raises a critical pedagogical concern: students might rely heavily on tools like Copilot to complete assignments without genuinely understanding the underlying concepts. This challenge served as a key learning point during the development of Haskify. To address this issue, Haskify has been explicitly designed to avoid directly providing solutions. Unlike Copilot, the primary goal of Haskify is not to supply immediate answers, but rather to foster deep learning and comprehension. The AI assistant in Haskify offers guidance and hints while encouraging students to think critically and develop their own solutions—an educational philosophy that will be examined in greater detail later in this thesis.

2.1.3 ChatGPT-Powered Educational Tools

While GitHub Copilot and CS50 Duck Debugger are different in purpose and design, they share a common trait: both rely on OpenAI's large language models to assist learners in programming tasks. These tools demonstrate the versatility of ChatGPT when integrated into real-time coding environments. Their success and limitations motivated further exploration into ChatGPT's role in education, particularly around automated feedback and learning support—topics that have been investigated more formally in academic literature. One critical drawback observed from using ChatGPT in educational settings is its tendency to respond too quickly and provide full answers instantly. This speed, while impressive, can lead students to rely on answers without fully understanding the underlying concepts. As a result, learners may become passive recipients rather than active problem-solvers. This behavior contradicts the goals of effective tutoring, where reflection, trial and error, and step-by-step reasoning are crucial for deep learning. This insight influenced a core design principle in Haskify: the AI assistant avoids giving complete solutions outright. Instead, it guides users with hints and explanations that encourage comprehension and exploration. This balance between assistance and challenge is critical for fostering meaningful learning, particularly in functional programming, where conceptual clarity is essential.

2.1.4 Intelligent Tutoring Systems Combining APR and LLMs

Beyond real-time assistance through language models, a more integrated approach is presented by Fan et al. [4], who describe a multi-year effort to develop an Intelligent Tutoring System (ITS) for programming education. This ITS combines automated program repair (APR) techniques with large language models (LLMs) to offer precise feedback on student code submissions. The system operates by detecting errors, repairing them, and then using LLMs to explain those fixes in natural language. One of the strengths of this ITS is its hybrid architecture: low-level repairs are generated via static and dynamic program analysis, while high-level feedback explanations are handled by LLMs. This modular setup is language-independent and extensible, allowing the system to scale across different programming courses and be continuously improved by software engineering students as part of a long-term educational project. While the ITS emphasizes correction and explanation, it raises an important pedagogical consideration: giving too much help too early can bypass learning. In contrast, Haskify does not apply automated repairs; instead, it prioritizes human-like tutoring through GPT-based interaction, designed to guide students toward understanding without directly solving the problems for them. This philosophical difference ensures that students using Haskify remain engaged in the problem-solving process, which is particularly crucial in learning abstract functional programming concepts. Thus, while both systems seek to scale tutoring and feedback, Haskify's emphasis lies in fostering active learning through explanation, not automation.

2.2 ChatGPT in Educational Research

2.2.1 Automated Feedback for Open-Ended Tasks

A particularly relevant contribution to the domain of AI-supported education is presented by Dai et al. [2], who investigated the potential of ChatGPT to generate feedback for student assignments. Their study focused on assessing the *readability*, *alignment with instructor feedback*, and *effectiveness* of feedback generated by ChatGPT in the context of open-ended tasks, specifically project proposals in a data science course. The researchers analyzed 103 student proposals and corresponding instructor comments, prompting ChatGPT to generate feedback along five assessment dimensions: project goals, topic appropriateness, business benefit, novelty, and clarity. Remarkably, the feedback produced by ChatGPT was rated as more readable and coherent than that of human instructors, as determined through a five-point expert rating scale. This finding highlights the potential of language models to deliver polished and accessible educational support at scale. The study also examined whether ChatGPT's feed-

back agreed with that of instructors, focusing on the polarity (positive, negative, or none) of feedback across different assessment aspects. Results indicated strong agreement in some categories (e.g., *topic*), but weaker precision in others (e.g., *goal* and *benefit*), suggesting limitations in the model's reliability for performance assessment. Another interesting observation was ChatGPT's unexpected capacity to generate *process-focused* feedback—guidance on how to approach a task—alongside task-level evaluations. Although ChatGPT did not provide feedback at the self-regulatory or personal (self) level, its ability to generate strategy-oriented comments positions it as a promising tool for formative feedback. Overall, this study underscores both the capabilities and limitations of ChatGPT in educational contexts. It informs the development of tools like Haskify by reinforcing the importance of balancing helpfulness with pedagogical integrity—ensuring that learners are guided rather than given full answers, a principle Haskify is designed to uphold. Having examined a range of AI-assisted educational tools and tutoring systems, it becomes clear that Haskify addresses a unique niche by combining domain-specific guidance in functional programming with pedagogically conscious feedback. In the following chapter, we detail the system requirements and design choices that shaped Haskify's development. These decisions were guided both by insights gained from the literature and the practical challenges of supporting Haskell learners in an interactive, scalable, and ethical way.

3 System Requirements and Design

3.1 Requirements Analysis

The goal of this section is to identify and define the needs and expectations of the users of the Haskify. Haskify is a web-based educational tool designed to support students learning functional programming through Haskell. It integrates an AI assistant, a Haskell code editor with a live compiler.

Primary Users:

- **Students:** Interact with the system to write code, receive feedback, and learn.
- **Educators:** Could review usage or adapt exercises in future extensions.

System Objectives:

- Provide a web-based Haskell code editor with syntax highlighting and compilation.
- Deliver AI-driven guidance and explanation for functional programming exercises.
- Enhance the learning experience through immediate feedback.

3.2 Functional Requirements

FR1: The system shall allow users to write and edit Haskell code in the code editor.

FR2: The system shall compile and execute Haskell code and display the output or errors.

FR3: The system shall provide an AI assistant that accepts user questions and code-related queries.

FR4: The AI assistant shall respond with educational hints or explanations related to Haskell and functional programming.

FR5: The system shall detect and highlight syntax or runtime errors in user code.

3.3 Non-Functional Requirements

NFR1: **Usability:** The interface should be intuitive and accessible to first-year students.

NFR2: **Performance:** The code compilation and AI responses must complete within 2 seconds under normal load.

NFR3: **Availability:** The system should be available 24/7 with minimal downtime.

NFR4: **Scalability:** The system should support at least 100 concurrent users during peak times.

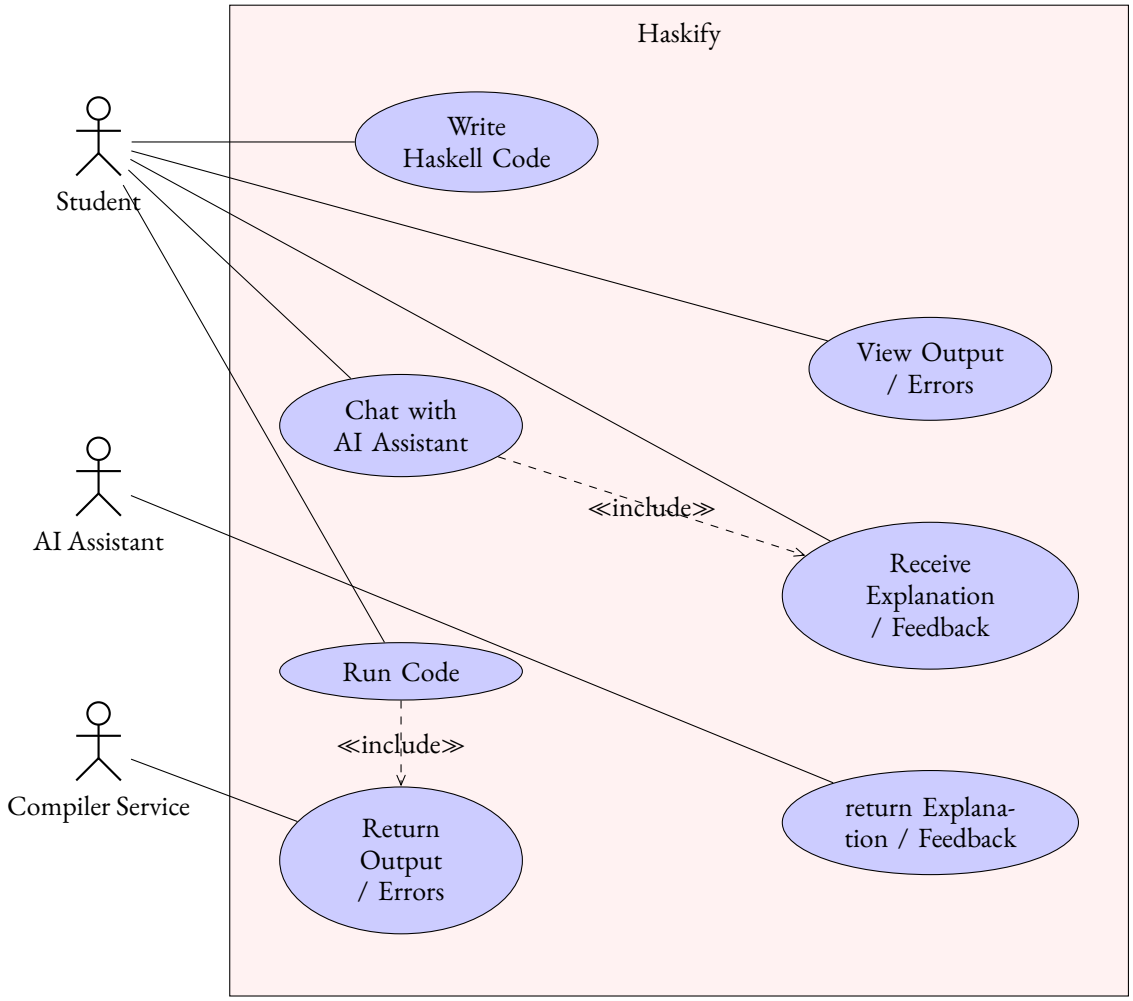
NFR5: **Maintainability:** The codebase should be modular and follow good software engineering practices.

NFR6: **Portability:** The app must run in all major browsers (Chrome, Firefox, Safari).

These requirements were considered throughout the development process. Usability was prioritized by selecting the Monaco Editor, giving students a coding experience similar to Visual Studio Code. Performance was addressed by keeping the AI assistant's prompt context small and by optimizing the GHC compilation sandbox. We avoided heavy UI frameworks to keep load times low and interactions snappy. To meet availability and scalability goals, we designed stateless HTTP APIs and deployed the system as follows: the Dockerized Node.js backend runs on Render, and the React/Vite frontend is built and served on Vercel's global CDN. Portability was verified through cross-browser testing (Chrome, Firefox, Safari). These constraints directly shaped our design and technology choices (see §3.5.4).

3.4 Use Case Diagram

Below is the use case diagram for Haskify



Use Case Specifications

Use Case: Write Haskell Code

- **Actor:** Student
- **Description:** The student writes Haskell code in the integrated editor.
- **Preconditions:** The user has opened the editor interface.
- **Postconditions:** The code is stored in memory and ready to be compiled.

Use Case: Run Code

- **Actor:** Student
- **Description:** The user triggers code execution. The code is sent to the backend compiler.
- **Preconditions:** Valid Haskell code is present in the editor.
- **Postconditions:** Output or errors are returned and displayed.

Use Case: Chat with AI Assistant

- **Actor:** Student
- **Description:** The student sends a query about Haskell code or concepts.
- **Preconditions:** An AI session is active.
- **Postconditions:** A helpful, domain-specific response is displayed.

Use Case: Receive Explanation / Feedback

- **Actor:** AI Assistant
- **Description:** The assistant analyzes context and provides an explanation, hint, or guidance.
- **Preconditions:** A relevant user query was received.
- **Postconditions:** A response is generated and sent to the user.

Use Case: Return Output / Errors

- **Actor:** Compiler Service
- **Description:** Executes code and sends output or error information.
- **Preconditions:** Valid compile request received.
- **Postconditions:** Result is returned to frontend for display.

3.5 Technology Stack

Haskify is implemented using modern web technologies combined with cloud-based AI and compiler integration. The system is structured into a frontend and backend architecture to ensure scalability and maintainability.

3.5.1 Frontend

- **React.js:** Used to build the user interface with component-based design.
- **Vite:** A fast development server and build tool for modern JavaScript applications.
- **HTML/CSS:** Used for layout and styling, including responsive design.
- **Monaco Editor:** Embedded code editor with Haskell syntax highlighting and real-time editing. We use Monaco with a Haskell Monarch tokenizer [1].

3.5.2 Backend

- **Node.js:** JavaScript runtime used to build the backend server and API routes.
- **Express.js:** Backend web framework for handling API requests.
- **Nodemailer:** Handles email functionality (e.g., contact form).
- **Haskell (GHCi):** Used for compiling and running submitted Haskell code in a sandboxed environment.

3.5.3 AI Assistant

- **DeepSeek API:** Powers the AI assistant with a specialized focus on functional programming guidance.
- **Custom prompt engineering:** Ensures AI answers remain educational and within domain scope.

3.5.4 Other Tools and Services

- **GitHub:** Version control and collaborative code management.
- **Render:** Hosts the Dockerized Node.js backend with automatic builds and zero-downtime updates.
- **Vercel:** Deploys the React/Vite frontend via Git integration to a global CDN.

4 Implementation

4.1 System Architecture

The architecture of *Haskify* is designed to support interactive learning in functional programming by combining real-time Haskell code execution and AI-assisted tutoring within a unified web platform. The system is divided into two main layers: the **Frontend** (user-facing interface) and the **Backend** (logic and service orchestration). Figure 4.1 shows the complete architectural overview of the platform.

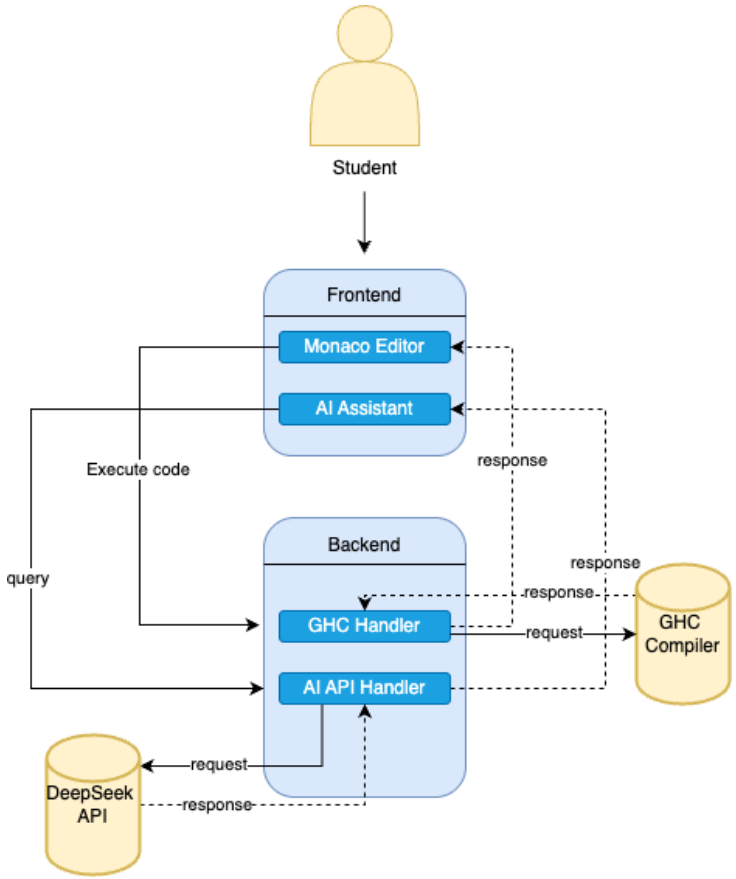


Figure 4.1: System Architecture of Haskify

Frontend

The frontend is implemented using React and Vite, and includes two key components:

- **Monaco Editor**: Provides an in-browser environment for writing Haskell code. It uses a custom Haskell syntax definition via a Monaco tokenizer (see Section 4.2). When the student executes the code, it sends a POST request to the backend endpoint `/run-haskell`, including both the code and any user input.
- **AI Assistant**: A conversational interface that enables students to ask questions about their Haskell code, output errors, or theoretical concepts. It communicates with the backend via the endpoint `/ai/ask`, forwarding the current code context and output to enrich the AI's response.

The frontend acts as a bridge between the user and backend services. It also manages state across components to synchronize the editor content and assistant conversation context.

Backend

The backend is built with Node.js using the Express framework. It exposes RESTful endpoints for handling code execution and AI queries. It consists of two core handlers:

- **GHC Handler (`/run-haskell`)**: Responsible for compiling and running submitted Haskell code. It creates a temporary `.hs` file, invokes GHC via a child process, and captures standard output or errors. Unsafe modules such as `System.IO.Unsafe` are blocked server-side to maintain security.
- **AI API Handler (`/ai/ask`)**: Receives the student's query, code, and program output, and constructs a prompt which is forwarded to an external AI provider (DeepSeek) using the OpenAI-compatible API. The prompt structure ensures domain-specificity, avoids giving direct answers, and encourages reflective guidance.

The backend is also responsible for rate-limiting, error formatting, and prompt safety — ensuring both functionality and academic integrity.

External Services

Two core external services are integrated into the backend logic:

- **GHC Compiler**: The Glasgow Haskell Compiler is used to execute code. It is called from within the backend using the `child_process` module, ensuring sandboxed and time-limited execution.

- **DeepSeek API**: A hosted large language model API, compatible with OpenAI's chat endpoint. The backend uses this API to generate responses from the assistant based on structured prompts.

Overall, the architecture reflects a clear separation of concerns between user interface, application logic, and external services, ensuring that both Haskell execution and AI support remain modular, scalable, and maintainable.

4.2 Frontend Implementation

Overview

The frontend of Haskify is built as a modern, single-page web application using React and Vite. The initial user interface design was created using Figma, allowing rapid prototyping and iteration before implementation in React. This helped establish the visual layout, component structure, and responsive behavior of the application. It provides a clean and responsive interface that allows students to write Haskell code, execute it, and interact with an AI assistant—all within the same environment. The component-based structure of React is leveraged to encapsulate functionality into logical units, promoting reusability and maintainability. The layout consists of a dual-pane grid: on the left, the Haskell code editor; on the right, the AI assistant interface. State is shared across components using top-level React state variables passed down via props, ensuring the AI assistant has real-time access to the current code and output state.

Monaco Editor Integration

To provide an in-browser environment for writing and running Haskell code, Haskify uses the Monaco Editor—the same engine behind Visual Studio Code. While Monaco does not natively support Haskell, it provides extensibility via custom language definitions. To add Haskell syntax highlighting, we incorporated a custom tokenizer based on an open-source project. This tokenizer was defined in the file `monaco-haskell.js` and registered at runtime using Monaco's `setMonarchTokensProvider` and `setLanguageConfiguration` methods. This enabled keyword recognition, comment handling, and basic type highlighting for common Haskell constructs. The component `HaskellEditor.jsx` wraps the Monaco editor and exposes additional features such as:

- Highlighting changed lines using `deltaDecorations`.
- Real-time output display for execution results.

- An input field for supplying runtime input to the program.
- A `Run` button which triggers a POST request to the `/run-haskell` endpoint.

Code execution results are captured and displayed directly below the editor using a styled `<pre>` element that emulates a terminal output.

AI Assistant Component

The AI assistant is implemented in `AIAssistant.jsx` and provides a chatbot-like interface for interacting with a domain-specific language model. The assistant supports the following features:

- Typing animation and initial greeting.
- Chat history displayed in a scrollable container.
- Syntax-highlighted responses, including code blocks.
- One-click *"Apply Code"* buttons to transfer suggested code into the editor.

Internally, user queries are sent to the backend endpoint `/ai/ask`, along with the current code and last output. The backend constructs a prompt and forwards it to the DeepSeek API, which returns an educational, domain-specific answer. Responses are parsed and rendered with Markdown-like formatting using `react-syntax-highlighter`.

Layout and UI Design

The layout is managed using CSS Grid and Flexbox, with styles modularized using CSS files co-located with each component. The top-level layout uses a grid with two main columns, each housing one of the key components (Editor and Assistant). Responsive styling ensures accessibility on smaller viewports. Additional components like the `Header`, `Footer`, and `ContactModal` contribute to usability without overwhelming the user.

State Management and Component Communication

The entire application is coordinated through a shared state object initialized in `App.jsx`. It holds:

- `code`: The current code in the editor.
- `output`: The latest execution result.

Updates to this state are passed to both the Monaco Editor and the AI Assistant via props, ensuring synchronization between what the user sees and what the assistant responds to. This shared-state approach enables tight coupling between functionality without introducing external state management libraries, maintaining simplicity and control within the React component model.

4.3 Backend Implementation

Overview

The backend of Haskify is implemented using Node.js and the Express framework. It is designed to handle both code execution requests and AI assistant queries through a modular and stateless REST API. Each functionality is exposed through its own endpoint, enabling a clear separation of responsibilities and easier maintainability. The backend performs three core roles:

- Handling Haskell code compilation and execution using GHC.
- Processing user questions by forwarding them to a domain-specific AI model (DeepSeek).
- Supporting auxiliary features such as contact form messaging and system health checks.

All communication between the frontend and backend occurs via HTTP POST requests, with responses returned as structured JSON. The backend also includes input validation, rate limiting, and prompt filtering to ensure safe and controlled interaction.

`/run-haskell` Endpoint and GHC Integration

The `/run-haskell` endpoint accepts a JSON object containing Haskell code and optional user input. It performs the following steps:

1. Writes the Haskell code to a temporary `.hs` file in the `/tmp` directory.
2. Compiles the file using the Glasgow Haskell Compiler (GHC) via Node's `child_process.exec`.
3. If compilation is successful, the program is executed with the provided input using a shell pipe.
4. The output (or any errors) is captured and returned to the frontend.

To maintain execution safety, the backend blocks code containing unsafe modules such as `System.IO.Unsafe` or uses of `unsafePerformIO`. A timeout is also applied during execution to prevent long-running or infinite loops. Temporary files (`.hs`, `.out`, `.hi`, `.o`) are deleted after each run to avoid resource leaks.

`/ai/ask` Endpoint and Prompt Logic

The `/ai/ask` endpoint powers the AI assistant component. It accepts three parameters: the user's query, the current Haskell code, and the last execution output. The backend then assembles a structured system prompt that includes:

- Instructions to answer like a Haskell tutor: concise, focused, and non-repetitive.
- A reference to the user's current code and program output.
- Optional example prompts to guide the model's tone and structure.

This prompt is forwarded to DeepSeek via its OpenAI-compatible API method. The response is streamed and returned to the frontend for display in the AI chat. The model used is configured to avoid giving direct solutions and instead provide educational feedback, including code hints, syntax corrections, and theory reminders.

Security, Rate Limiting, and Input Handling

To ensure stable operation and prevent abuse, the backend includes several safety mechanisms:

- **Rate Limiting:** Express Rate Limit is used to cap requests per IP over a defined window, especially for code execution.
- **Input Size Limits:** Code exceeding 10,000 characters is rejected to prevent memory overload.
- **Error Handling:** All endpoint handlers catch and return structured error messages with relevant HTTP status codes.
- **Prompt Sanitization:** The prompt generation filters unsupported content and focuses on pedagogically sound guidance.

These features collectively ensure that users cannot exploit the backend or overload the system, and that responses remain within the educational scope of the platform.

Contact Form and Health Check Endpoints

The backend exposes two additional endpoints:

- `/api/contact`: Handles form submissions from the frontend's contact modal. Messages are sent to a predefined email address using Nodemailer with Gmail as the transport.
- `/health`: A lightweight GET endpoint that returns a simple `200 OK` if the server is online. This is used by the frontend on load to verify backend availability.

These endpoints add minor but essential supporting functionality to the system.

5 Evaluation and Experiment Design

5.1 Overview and Research Questions

We conducted a usability-oriented evaluation of **Haskify** to assess its perceived usefulness and intuitiveness for supporting Haskell learning and exam preparation.
RQ1: Does Haskify help novices complete Haskell tasks successfully?
RQ2: Does Haskify reduce perceived cognitive load and increase perceived understanding?

5.2 Participants and Ethics

We recruited 23 first-year CS students enrolled in the "Programming Paradigms and Compiler Construction" (PPDC) course at Goethe University Frankfurt. Participation was voluntary and anonymous, with informed consent obtained via an online form prior to the study. No personal data beyond self-reported prior experience and confidence in Haskell was collected. All participants had taken or were currently taking the PPDC course, ensuring relevant background knowledge.

5.3 Tasks and Materials

Participants completed four short tasks inside Haskify:

1. **Fill in the blanks (even numbers filter):** complete a Haskell list comprehension that returns only even numbers from a list.

2. **AI-generated practice:** ask the AI assistant to propose a basic Haskell exercise for functional programming exam practice, then solve it.

3. **List comprehension output prediction:** predict the output of a given list comprehension before testing it in the editor.

Participants were encouraged to ask the AI assistant for help when needed and to compile their solutions in the integrated editor. We provided a short instruction sheet and a post-task feedback form (Google Form).

5.4 Study Design

The study followed a single-condition design where all participants used Haskify for the assigned tasks. This format was chosen to focus on user perceptions and direct task performance without introducing baseline variation from alternate tools. The total duration was approximately *15 minutes*: *10 minutes* for tasks and *5 minutes* for the feedback form.

5.5 Measures and Instruments

We captured:

- **Effectiveness:** task success (Yes/No/Partial) based on self-report and observed completion.
- **Perception:** ease-of-use (1–4 scale), self-reported confidence, perceived understanding (Yes/No), and intention to reuse.
- **Behavioral:** whether participants interacted with the AI assistant and compiled their code during the session.

5.6 Results

5.6.1 Participant Background

All 23 participants (100%) had taken or were currently enrolled in the PPDC course. Self-reported Haskell confidence levels before the experiment were: 34.8% "Not confident", 43.5% "Somewhat confident", 17.4% "Confident", and 4.3% "Very confident".

5.6.2 Effectiveness

Across all four tasks, 73.9% of participants fully completed the exercises, 21.7% partially completed them, and 4.3% did not complete the tasks. Participants frequently used the AI assistant to resolve syntax or logic errors, particularly in the "Fill in the blanks" and AI-generated practice tasks.

5.6.3 Perceived Understanding

A strong majority (87.0%) reported that using Haskify helped them better understand the problems they worked on, while 13.0% indicated no significant difference.

5.6.4 Ease of Use and User Experience

Ease-of-use ratings were high: 43.5% rated the app at 4/4, 34.8% at 3/4, and 21.7% at 2/4; no participants selected 1/4. Most participants (91.3%) expressed an intention to use Haskify again for Haskell learning or exam preparation.

5.6.5 Summary

Overall, the results suggest that Haskify was perceived as easy to use, improved task completion rates, and enhanced conceptual understanding for most participants. These findings support both RQ1 and RQ2, indicating that an AI-supported, domain-specific coding environment can positively influence novice learners' functional programming experience.

6 Didactic Reflection

This chapter aims to explore the pedagogical value of Haskify by reflecting on established educational theories and frameworks, highlighting how Haskify supports effective learning in the context of functional programming, specifically Haskell.

6.1 Educational Objectives and the Role of Haskify

Functional programming, particularly with Haskell, presents unique challenges to students, including abstract reasoning, symbolic literacy, and understanding complex hierarchical structures [6]. According to Li et al. [6, p. 1], algebra and abstract reasoning introduce significant challenges because they require students to adapt prior knowledge into new, more abstract frameworks. Haskify addresses these educational challenges by fostering deeper understanding and engagement through its carefully structured learning environment. It provides immediate, context-sensitive feedback and promotes active, exploratory learning rather than passive knowledge absorption.

6.2 Pedagogical Foundations of Haskify

Several established pedagogical theories underpin Haskify's design: **Constructivism** posits that learners actively construct knowledge through experience and exploration. Haskify supports constructivist principles by allowing students to experiment freely with code in a risk-free environment, encouraging iterative refinement of understanding through trial and error. **Scaffolding**, inspired by Vygotsky's Zone of Proximal Development (ZPD), emphasizes tailored support to bridge the gap between what learners can achieve independently and what they can achieve with guidance. Li et al. [6, p. 4] note that intelligent tutoring systems (ITS), which provide "personalized feedback and guidance," significantly enhance learning by aligning support with learners' immediate needs. Haskify's AI assistant embodies this scaffolding approach. Rather than offering direct solutions, it nudges students towards understanding through hints, guiding questions, and explanations, thus supporting learners in gradually developing independence in their coding practices.

6.3 Cognitive Load and Concretization

Cognitive Load Theory suggests that learning effectiveness improves when instructional design accounts for the learner's cognitive processing capacity. Li et al. [6, p. 4] highlight concretization—making abstract problems tangible—as effective in reducing cognitive load and improving conceptual understanding. In Haskify, abstract functional programming concepts are concretized through explicit, context-sensitive error messages provided by the integrated real-time compiler, and detailed explanations from the AI assistant. These features clarify abstract concepts by directly linking theoretical constructs with tangible, executable code examples.

6.4 Comparative Advantages of Haskify

Compared to traditional lecture-based or static tutorial resources, Haskify offers superior pedagogical value in several key areas:

- **Personalized Learning**: Feedback adapts based on each learner's interaction history, tailoring responses to support individual learning journeys.
- **Real-Time Interaction**: Immediate feedback promotes continuous learning loops, crucial for mastering complex, incremental concepts.
- **Emotional and Motivational Support**: Haskify's AI assistant provides socio-emotional encouragement, positively influencing students' motivation and persistence in tackling challenging concepts [6, p. 11].

6.5 Conclusion

In summary, Haskify closely aligns with robust educational theories and research-based best practices for effective learning. By integrating constructivist principles, scaffolding, and cognitive load management, Haskify represents a significant advancement in functional programming education, potentially transforming how learners engage with abstract programming concepts.

7 Conclusion & Future Work

7.1 Summary of Contributions

This thesis presented **Haskify**, a web-based learning environment for functional programming in Haskell that combines a Monaco-based code editor, a server-side GHC compilation pipeline, and a domain-specialized AI assistant (DeepSeek API, OpenAI-compatible). The work contributes:

- **System design and implementation:** An end-to-end architecture that integrates a syntax-aware editor, real-time compilation via GHC, and an AI assistant designed to scaffold rather than solve.
- **Pedagogically aligned tutoring strategy:** Prompting and guardrails that support a *scaffold-not-solve* philosophy consistent with constructivist learning and didactic reflection discussed in this thesis.
- **Haskell editing support in Monaco:** Adoption and integration of a Haskell Monarch tokenizer to provide usable syntax highlighting and editor affordances in a setting where Monaco has no first-party Haskell support.
- **Deployable, reproducible setup:** A Dockerized backend on Render and a frontend deployed on Vercel, enabling reproducible builds and easy sharing.
- **Planned empirical evaluation:** A concrete experiment design (tasks, measures, instruments, and analysis plan) to assess effectiveness and cognitive load against a static tutorial baseline.

7.2 Limitations

This work has several practical and methodological limitations. First, recruiting participants and motivating them to use Haskify outside of regular coursework proved challenging, which may limit sample size and generalizability. Second, the system currently relies on free-tier or low-cost hosting, where network and cold-start latencies can vary and confound perceived

performance. Third, the task set focuses on introductory Haskell topics (e.g., pattern matching, list comprehensions), which may not capture deeper conceptual hurdles (e.g., monads, type classes in larger code bases). Fourth, Monaco lacks native Haskell support; relying on a community tokenizer entails gaps (e.g., edge-case highlighting) that could affect user experience. Finally, dependence on a third-party LLM API introduces version drift and availability risks.

7.3 Future Work

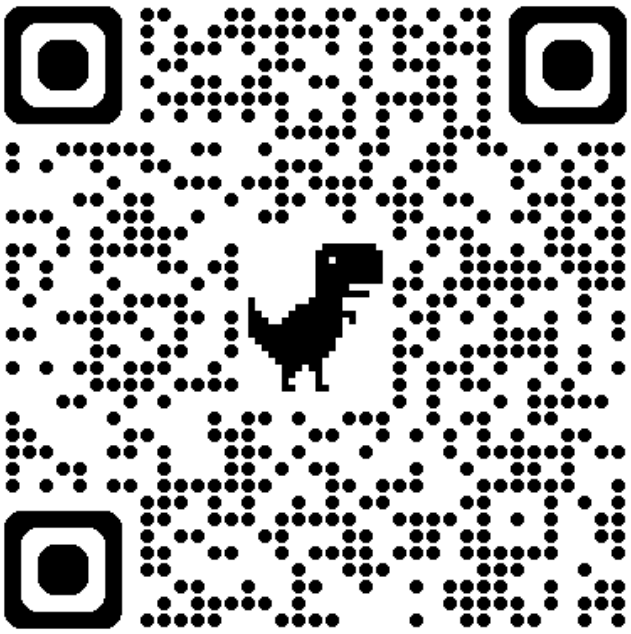
Building on this foundation, we propose:

1. **Complete and scale the evaluation:** Run the planned study with a larger cohort, report results, and add follow-up/retention measures.
2. **Richer hint taxonomy:** Map common novice error categories to graded hint types (pointer, Socratic question, micro-example) and evaluate which combinations are most effective.
3. **Expanded exercise coverage:** Add tasks on recursion patterns, higher-order functions, algebraic data types, and (carefully) introductory monads.
4. **Instructor tooling:** A lightweight dashboard for viewing anonymized progress and common misconceptions to support course integration.
5. **Performance & robustness:** Warm starts, caching, and sandbox hardening for GHC; improved latency on the assistant side.
6. **Comparative baselines (optional extension):** A compact comparison against a general LLM tutor to isolate the value of domain-specific guardrails.
7. **Accessibility and internationalization:** Keyboard-only workflows, screen-reader labels, and localized prompts.

Taken together, these next steps aim to strengthen both the empirical evidence for Haskify's pedagogical value and the robustness of the system for classroom use. For source code and study materials, see the *Artifacts & Code Availability* page.

Artifacts & Code Availability

Repository: https://github.com/Ahmadkhdeir/haskify
This repository contains the full source code (frontend and backend) and the Dockerfile.



Scan to open the project repository.

Bibliography

[1] byteally. monaco-haskell: Haskell highlighting for monaco editor. https://github.com/byteally/monaco-haskell.

[2] Wei Dai, Jionghao Lin, Hua Jin, Tongguang Li, Yi-Shan Tsai, Dragan Gašević, and Guanliang Chen. Can large language models provide feedback to students? a case study on chatgpt. In *2023 IEEE international conference on advanced learning technologies (ICALT)*, pages 323–325. IEEE, 2023.

[3] SRM Derus and AZ Mohamad Ali. Difficulties in learning programming: Views of students. In *1st International Conference on Current Issues in Education (ICCIE 2012)*, pages 74–79, 2012.

[4] Zhiyu Fan, Yannic Noller, Ashish Dandekar, and Abhik Roychoudhury. Software engineering educational experience in building an intelligent tutoring system. *arXiv preprint arXiv:2310.05472*, 2024.

[5] Stef Joosten, Klaas Van Den Berg, and Gerrit Van Der Hoeven. Teaching functional programming to first-year students. *Journal of Functional Programming*, 3(1):49–65, 1993.

[6] Chenglu Li, Wanli Xing, Yukyeong Song, and Bailing Lyu. Rice algebrabot: Lessons learned from designing and developing responsible conversational ai using induction, concretization, and exemplification to support algebra learning. *Computers and Education: Artificial Intelligence*, 8:100338, 2025.

[7] Rongxin Liu, Carter Zenke, Charlie Liu, Andrew Holmes, Patrick Thornton, and David J Malan. Teaching cs50 with ai: leveraging generative artificial intelligence in computer science education. In *Proceedings of the 55th ACM technical symposium on computer science education V. 1*, pages 750–756, 2024.

[8] Iain Milne and Glenn Rowe. Difficulties in learning and teaching programming—views of students and tutors. *Education and Information technologies*, 7:55–66, 2002.

[9] Ben Puryear and Gina Sprint. Github copilot in the classroom: learning to code with ai assistance. *Journal of Computing Sciences in Colleges*, 38(1):37–47, 2022.