In this article, we describe PrairieLearn, a flexible open-source platform for asking questions to students that is in broad use for both homework and exams. We demonstrate PrairieLearn’s flexibility and ability to integrate existing code and questions into a single platform using three case studies: Parson’s problems, designing finite-state machines, and auto-grading “Explain in plain English” questions. We highlight aspects of PrairieLearn’s structure that enable this flexibility, in particular PrairieLearn’s ability to execute arbitrary code both during the generation of a question instance and during grading student answers.

CCS Concepts
• Software and its engineering → Software organization and properties; • Social and professional topics → Student assessment;

Author Keywords
assessment, software architecture, flexible, extensible

INTRODUCTION
In the past two decades, we’ve seen significant growth in the number and usage of web-based platforms for asking questions related to course content. In addition to traditional Learning Management Systems (LMS, e.g., Canvas, Moodle), there have been publisher solutions that provide content associated with textbooks (e.g., WileyPlus, Connect), start-ups focused on providing domain-specific commercial platforms (e.g., Codio, Vocareum), as well as a wide range of solutions that instructors have built for themselves. Because they can potentially be used for both homework and exams, we refer to these as question-asking platforms (QAPs).

In most STEM classes, a large amount of content is objectively gradeable, enabling QAPs to perform auto-grading. Auto-grading provides the pedagogical benefit of immediate feedback to students, while simultaneously reducing the workload of instructors. Using auto-grading—where it is warranted—enables instructors and other course staff to focus their efforts on higher value tasks, like tutoring and grading subjective content (e.g., coding style).

For all that QAPs offer, we find that the current QAP landscape largely presents two options to instructors: (a) QAPs that enable easy authoring of a relatively small number of question types, or (b) the freedom to implement questions however they want by writing a new platform from scratch. This paper focuses on a software architecture designed to (c) provide significant flexibility while retaining the ability to easily implement common question features.

Figure 1. The current question-asking platform (QAP) landscape largely presents two options to instructors: (a) QAPs that enable easy authoring of a relatively small number of question types, or (b) the freedom to implement questions however they want by writing a new platform from scratch. This paper focuses on a software architecture designed to (c) provide significant flexibility while retaining the ability to easily implement common question features.
unsatisfied with these commercial offerings build their own specialized tools, represented by Figure 1(b). While interoperability standards like LTI [16] and SCORM [31, 4] are useful for integrating a collection of specialized tools into a single interface, they don’t decrease the effort of implementing specialized tools.

This paper describes PrairieLearn [39, 38], an open-source question asking platform that provides both flexibility to author novel question types (approaching what could be obtained from a custom platform) and ease to author common question types (approaching what could be obtained from a commercial QAP). PrairieLearn is actively being used by the instructors of over 100 courses across multiple universities in a broad range of subjects (e.g., CS, Mechanical Eng., Chemistry, Statistics, Nutrition) and typically grades over 100,000 student answers daily. In addition, PrairieLearn has been used in a number of research studies [1, 2, 3, 5, 7, 9, 10, 8, 6, 12, 14, 23, 24, 27, 30, 32, 33, 28, 34, 35] and was a critical to the development of the Computer-Based Testing Facility (CBTF) [40, 41, 42, 43].

In Section 2, we attempt to demonstrate PrairieLearn’s flexibility by showing a range of example questions that it supports. In Section 3, we highlight the three principles of PrairieLearn’s design to which we attribute this flexibility, specifically:

1. Use of standard web technologies (Section 3.1)
2. Encapsulation of common functionality as Elements (Section 3.2)
3. Support for external auto-graders (Section 3.3)

DEMONSTRATION OF FLEXIBILITY

We include images and descriptions of six problems that exemplify the kind of flexibility that PrairieLearn provides. For this audience, we’ve chosen to focus on CS-related questions, but there are similar questions in other domains including 3D-manipulation of robots, drawing forces to complete free-body diagrams, and randomly parameterized truss structures. Importantly, none of these question types are directly supported by the tool; these questions have been implemented on top of the interfaces that the tool provides.

In addition to multiple-choice questions, one bread-and-butter type of question is a code writing question (Figure 2(a)). This kind of question builds on the text editing element (Section 3.2) and the external auto-grader (Section 3.3) to allow an instructor-defined set of tests and scoring rubric to be applied to the student code.

A QAP is much more useful if it supports the development of *item generators* [17], which generate random parameters in order to create one of a large collection of question instances. Item generators enable re-use of the same question each semester and mitigate cheating because each student receives a different version of the question. Figure 2(b) shows an example of such a generator that renders a random graph for the student and asks the student to determine a feature of the graph. Figure 2(c) shows another item generator that asks students to compute the values on datapath and control wires of a MIPS processor for a randomly generated instruction.

In the sub-sections that follow, we provide detailed discussion of three problems that demonstrate PrairieLearn’s capabilities.
Parson’s Problems

Parson’s problems (shown in Figure 2(d)) present students with a collection of tiles containing lines of code; students are asked to select and order tiles to complete a program to accomplish a particular task. Parson’s problems were proposed to provide students an opportunity to practice problem solving that involved lower cognitive load and focused less on syntax [29].

The first Parson’s problem implementation in PrairieLearn was ported from the MIT licensed js-parsons library [18, 15] in one course’s repository. As PrairieLearn uses standard web technologies—see Section 3.1—the front end portion of code was largely untouched, with the tile randomization and grading restructured to fit PrairieLearn’s interfaces.

The code was implemented as a PrairieLearn element (Section 3.2). As an element, the Parson’s problem implementation is separated from the specific text associated with a given question. After a year’s use in one course, the element was significantly re-written to provide functionality beyond the original code; it now support both order-based correctness checking and running the constructed code against unit tests using the external autograder (Section 3.3) and released to all PrairieLearn users as the pl-order-block element.

Auto-grading Finite State Machines

Figure 2(e) shows a question that asks students to design a finite-state machine (FSM) with certain properties. Its implementation is similar to the Parson’s problem described in Section 2.1 in that its user interface is largely derived from existing open source code. Specifically, the mini CAD tool for drawing FSMs is also derived from MIT licensed software [37]. The interface allows creating new states, associating outputs with those states (only Moore machines are supported), connecting states with edges, labelling those edges with predicates, moving states and edges, and naming states (to help students keep track of their design).

When saved or graded, a textual representation of the student design is generated. This textual representation is auto-graded via simulation by running a collection of test vectors against the model and comparing the output to a “golden” implementation. Students are provided feedback by providing an example test input that failed and the series of outputs on their implementation and the expected output for that input.

Auto-grading ‘Explain in plain English’ questions

Figure 2(f) shows an implementation of an auto-grading Explain in plain English (EipE) question, where the student is shown a piece of code and asked to provide a natural language description of the code. The code reading skill, which EipE questions assess and provide practice for, is thought to be an important developmental skill in learning to program [19, 22, 20, 36, 26, 25]. Lister et al. state that, while their data doesn’t support the idea of a strict hierarchy, “We found that students who cannot trace code usually cannot explain code, and also that students who tend to perform reasonably well at code writing tasks have also usually acquired the ability to both trace code and explain code.”[21]

The implementation shown in PrairieLearn supports auto-grading the natural language descriptions of code that students provide using natural language processing (NLP) [11]. While PrairieLearn has no built-in support for NLP, it allows arbitrary libraries to be used by question code. In this case, we’re using Python’s natural language toolkit nltk to tokenize, unidecode, and pyspellchecker to do spelling correction. Then numpy is used to compute a score from the resulting bag-of-words and bigrams using a pre-trained model [11]. Because PrairieLearn allows authoring autograders in Python, our current implementation is built inside the questions themselves; if we wanted to use other languages, we could have implemented the autograder using the external autograder support.

FLEXIBILITY/EASE PRINCIPLES

Question Flexibility through an Authoring Interface using Standard Web Technologies

Many LMS’s provide highly structured authoring environments, often using a web-based GUI to fill in the content for an existing question format. While such an authoring environment meets its goal of simplicity and prevents bugs by construction, it also highly constrains the kinds of questions that can be asked.

In contrast, PrairieLearn enables questions to be authored using standard web technologies: HTML5 (including client-side Javascript), server-side Python, JSON for question metadata, and Docker for external autograders (see Section 3.3). This minimally constraining authoring interface has three important implications:

1. It facilitates instructors having all content under one tool.
2. It permits using existing software libraries for question construction (e.g., the questions shown in Figure 2(d-f)).
3. All content is stored in the git source code.

In practice, if you could build a stand-alone web site to ask your kind of question, you can implement your question using PrairieLearn. We’ve found this flexibility to be essential in our effort to move from paper exams to computer-based exams without dumbing down our courses (e.g., by only using multiple-choice questions). In fact, PrairieLearn facilitates novel question development because it provides authors with standard services (e.g., authentication, storing grades, logging) which allows authors to focus just on issues salient to the particular question.

Elements as Reusable Components

There are significant similarities among groups of questions. For some kinds of questions, the similarity is in the whole structure of the question (e.g., multiple-choice questions, Parson’s problems). In other questions, it is smaller components (e.g., accepting a numeric value with a particular precision, displaying a graph). It would be ridiculous to provide an authoring environment which requires question authors to replicate the functionality for these common components and structures.
While many questions can be auto-graded by elements or by deploying unreliable AI graders in high-transparency high-stakes exams. In *International Conference on Artificial Intelligence in Education*. Springer, 16–28.

**REFERENCES**


