UASSIGN: SCALABLE AND FLEXIBLE INTERACTIVE ACTIVITIES FOR TEACHING THE UNIX TERMINAL

BY

JACOB BAILEY

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2018

Urbana, Illinois

Adviser:

Associate Professor Craig Zilles

Abstract

In this thesis, I describe uAssign, an assignment system for teaching and assessing command line terminal skills. It allows an instructor to create terminal assignments where students complete a sequence of operations to achieve a goal correctly that can be automatically graded. Assignments can be randomized, and are created to an instructor's specification. uAssign provides a friendly user interface to a Docker container via an in-browser terminal emulator powered by WebSockets. Performance testing and its use in an ongoing course show that it is efficient enough to handle a large number of concurrent users. A survey of students shows significant improvement in terminal skill confidence after using uAssign and provides interesting insight into the preferences and opinions of freshman-level students on the subject of their development workflows and terminal experience.

Table of Contents

Chapter 1 Introduction

As computer science programs and other computing programs become more popular and more crowded, there is an increasing need to scale homework assignments and exams to support that need. At the University of Illinois, his is evidenced by new developments in the past few years such as Illinois' own PrairieLearn [23] and RELATE [16] to deliver online short-answer and coding assignments at scale, and the Computer-Based Testing Facility (CBTF) [24] [2] to administer computerized exams. These developments allow courses to scale their complicated assignments to handle the load while still keeping those assignments flexible and resistant to academic integrity violations.

However, one area that has not been given the same treatment is the subject of *terminal skills*. The terminal, or command-line, generally refers to the text-based interface that exists on nearly all computers. At a terminal, you have access to common utilities to perform actions on the system (file management, program execution) without using a graphical interface. A user can perform simple tasks, but also construct more complicated commands to do things that would be inefficient by hand. These sorts of skills are used heavily in the computer science curriculum and tech industry. These sort of skills are often relegated to course wiki pages and informal help from lab instructors, with assessment either non-existent or only through basic testing with multiple-choice quizzes. A discussion of related work can be found later in Chapter 9.

A solution to this problem is to offer terminal-based assignments alongside other course materials, allowing students to practice using the terminal and be assessed for proficiency. However, any implementation needs to meet both the expectations of students for their assignments and the expectations of instructors for a flexible and impactful platform.

1.1 Design goals

Ideally, terminal assignments should be consistent with students' other online assignments. In CS at Illinois, most lower-level courses use PrairieLearn, which offers many usability guarantees. To match, the terminal assignments should provide full access to a terminal in the browser, regardless of a student's own personal choice of operating system, browser, or location. Additionally, work should be saved to allow a student to return to an assignment without losing their progress, and be easily resettable if they'd like to start from scratch. Access to assignments should be as fast as possible and stay responsive as long as they keep the page open.

From the instructors' side, the platform should be flexible enough to support not only a wide range of terminal assignments, but to allow for them to be parameterizable and therefore unique for each student.

The terminal provided must be “real” enough to be representative of an actual machine, enabling students to carry their knowledge into other contexts. Secrets (like the answer to a specific version of an assignment) must be kept private from the student, known only to trusted parties.

To satisfy administrators, the platform must take security seriously, preventing abuse and resource exhaustion (intended or not). Meeting some of the previous goals may introduce attack vectors, so any implementation needs to ensure safety for all parties involved.

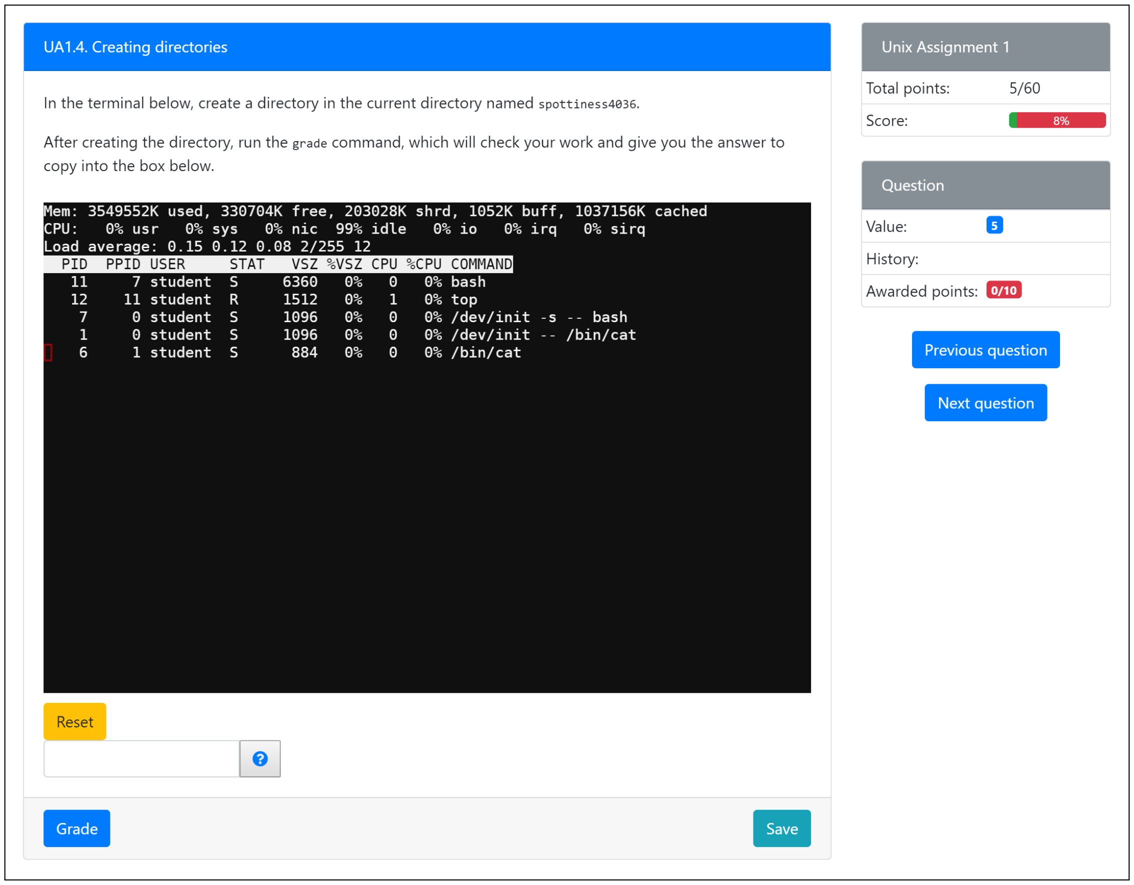
1.2 Contributions

This paper introduces *uAssign*, a system designed to administer terminal assignments by plugging into existing homework systems, allowing terminal skills to be practiced and assessed in an as-real-as-possible environment. uAssign aims to satisfy all of the aforementioned design goals, through the use of existing technologies such as Docker containers, WebSockets, an in-browser terminal emulator, and PrairieLearn.

Overall, a student will be able to open their browser, and be presented with a real, working terminal attached to a live system in which they can complete their assignment. This is not unlike other PrairieLearn questions with more complicated features (e.g. a finite state machine drawer or file uploader/editor), but tailored specifically for a terminal assignment. The interface as seen by a student can be seen in Figure 1.1.

Specifically, this thesis contributes the following:

1. The uAssign server, which manages container lifecycles (from creation to deletion).
2. A set of libraries for use in PrairieLearn, presenting a familiar experience to both students completing assignments and to instructors creating them.
3. A survey of students who have used uAssign and analysis of the results.
4. Performance testing.


UA1.4. Creating directories

In the terminal below, create a directory in the current directory named `spottiness4036`.

After creating the directory, run the `grade` command, which will check your work and give you the answer to copy into the box below.

```
Mem: 3549552K used, 330704K free, 203028K shrd, 1052K buff, 1037156K cached
CPU:   0% usr   0% sys   0% nic  99% idle   0% io   0% irq   0% sirq
Load average: 0.15 0.12 0.08 2/255 12
  PID  PPID USER     STAT   VSZ %VSZ CPU %CPU COMMAND
   11     7 student  S     6360   0%   0   0% bash
   12    11 student  R     1512   0%   1   0% top
    7     0 student  S     1096   0%   0   0% /dev/init -s -- bash
    1     0 student  S     1096   0%   0   0% /dev/init -- /bin/cat
    6     1 student  S      884   0%   0   0% /bin/cat
```

Figure 1.1: PrairieLearn terminal interface

Chapter 2 Definitions

In this section, we describe common terms used in this paper.

UNIX was an operating system created at Bell Labs, which spawned a family of similarly designed operating systems, including Linux, MacOS, Solaris, and others. The terms "UNIX-like" or "*NIX" are used more generally to refer to this family of operating systems to distinguish from the original UNIX operating systems.

UNIX is also associated with the "UNIX philosophy", which promotes writing programs that do one thing well and composing them together using text I/O to perform complicated tasks [18].

POSIX, or "Portable Operating System Interface" is a set of standards specified by the IEEE to allow for compatibility between UNIX-like operating systems [8]. It encompasses the API to the operating system (including process management, the C standard library, pipes, threads, etc), but also includes specifications for common terminal utilities such as `sh` (the POSIX shell), `grep`, `awk`, `ls`, and others. This cross compatibility both allows for the same tooling to be used on different systems, and for common skills to be transferred between different POSIX-compliant and mostly POSIX-compliant systems.

A **virtual machine**, or "VM", is an emulated machine running on another. A hypervisor emulates a full physical machine, including devices like storage drives and network cards, and can go as far as to emulate different processor architectures. Popular VM hypervisors include Xen, VirtualBox, and QEMU.

Although virtual machines are widely used and often make use of hardware acceleration to eliminate overhead, they have a non-trivial creation and startup time, which limits usability for one-off tasks.

A **container** is a general term for an isolated part of a system where a program (or set of programs) can run, with the perception of its own process tree, users, and filesystem. Containerization is implemented at the operating system kernel level, where containerized processes run alongside other processes on the same system. As such, containers have very little overhead compared to VMs, with near-instant startup time and resource impact on the host system. But, unlike VMs, containers are not flexible when it comes to details such as the kernel version and instruction set, inheriting those from the host.

Docker is a popular container platform which makes use of a kernel's isolation features [3]. Docker encompasses a daemon (`dockerd`) which manages containers, a command line client, and image registries like the Docker Hub. It has its own container image format, containing files varying from a single executable to a full operating system image. The Docker daemon exposes a REST API, allowing a client to build images, create and manage containers, as well as other container-related tasks. This interface makes Docker a good candidate as a part of another system which needs to work with containers directly.

A **Dockerfile** is a configuration file which Docker uses to build images [5]. It contains an ordered list of directives, which are performed on an intermediary container. Each directive performs a task, such as running a command or copying a file from the host. At each directive, a "layer" is created, which describes the differences from the previous layer. This final collection of layers becomes a Docker image, ready for distribution and use in creating containers.

PrairieLearn is an online learning and assessment system in common use at the University of Illinois [23]. It allows for flexible assignment creation using HTML, JavaScript, and Python. PrairieLearn is used both for online homework assignments and for exams in the CBTF [24] [2].

WebSocket is a bi-directional communication protocol, implemented by all major browsers [13]. It operates by "upgrading" a normal HTTP connection to a TCP connection, then communicating messages back and forth over a standardized protocol. Its design makes it simple to integrate into an HTTP server, and because a WebSocket takes control away from an HTTP connection, it too can make use of request URLs and headers, as well encryption by upgrading HTTPS to TLS.

A **UUID**, or universally unique identifier, is a 128-bit number which serves as an identifier [17]. The most commonly used type is UUIDv4, where 122 bits are randomly generated (the remaining 6 left to indicate the UUID's type). Due to their large size, UUIDs, have a statistically negligible collision rate, allowing multiple parties to generate their own UUIDs and feasibly never create a duplicate.

Chapter 3 High-level design

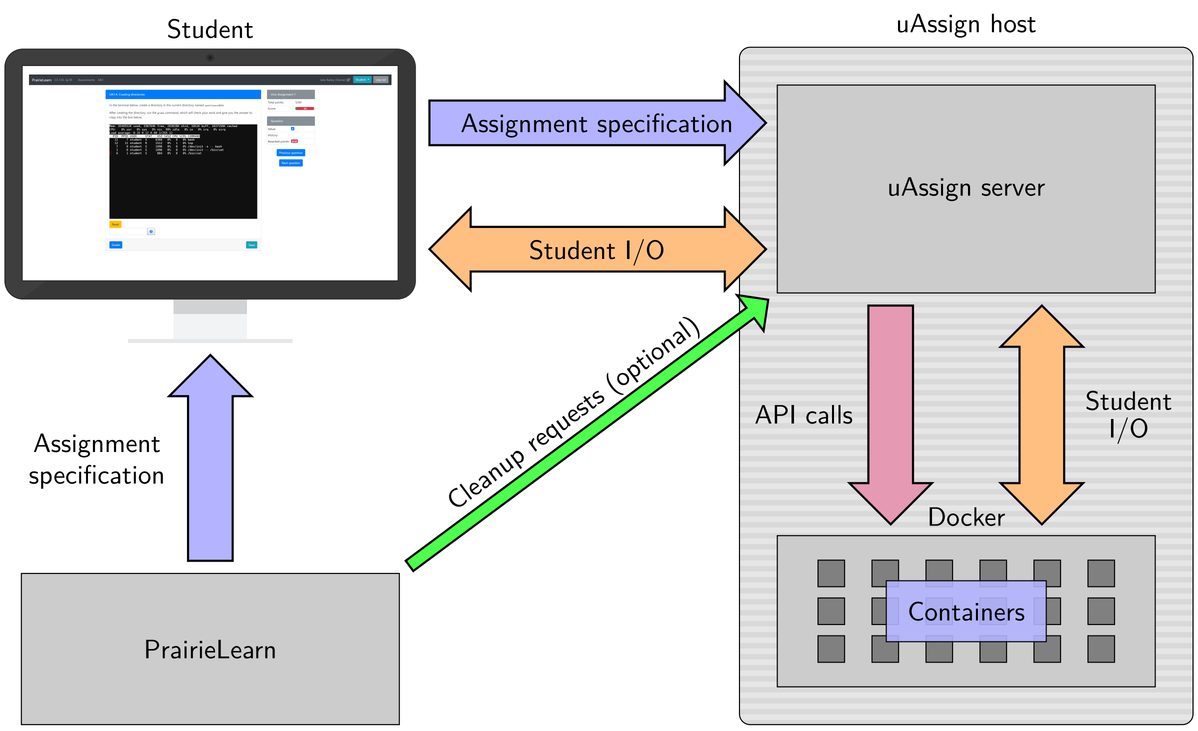


Figure 3.1: The overall design of uAssign.

The uAssign terminal assignment system is composed of a few relatively independent parts, illustrated in Figure 3.1:

- The uAssign server (shown on the right).
- A PrairieLearn server-side library (running in PrairieLearn, bottom left).
- A PrairieLearn client-side terminal interface (top left).

The uAssign server takes assignment specifications (the information needed to build an assignment), creates and manages the lifetimes of assignment instances, and performs maintenance with Docker to manage container resources. Chapter 5, Chapter 7, and Appendix A describe various duties the server performs.

The server-side library creates and encrypts assignment specifications. In this case, this is done in PrairieLearn as a part of its question generation. The generated specification is encrypted, and passed on to the student with the question as a parameter (similar to other PrairieLearn questions). Additionally, the server-side library allows PrairieLearn to send the uAssign server "cleanup requests", which instruct the uAssign server to preemptively cleanup any resources associated with an assignment specification. These are done when PrairieLearn knows a question is finished (on grade), and is done on a best-effort basis. Chapter 4 describes specifications and how the library is used.

The client-side terminal interface communicates the encrypted specification to the uAssign server, and displays a in-browser terminal interface for a student to complete an assignment. This is described in Chapter 6.

This design allows for specifications and their implementations to be separated. PrairieLearn produces a description of what it needs uAssign to produce, and code on the uAssign server implements that specification. This allows for bug fixes to occur on the server without needing to change PrairieLearn question generation (which in the case of potential exam use means a broken exam can be salvaged).

Chapter 4 Assignment specification

uAssign assignments begin as *specifications*. These specifications are the information sent to the uAssign server and used to create a container for use. The specification consists of a UUID, the assignment name (which is used by the server to locate files), as well an arbitrary blob of data.

Listing 4.1 shows an example specification for an assignment which asks a student to identify their current working directory. `specID` is a UUID which is unique to this specification. `assignmentName` is the name of the assignment as known to the server. Dots can be used to nest assignments together in directories, and the server uses this to locate files related to the assignment. `data` is an arbitrary JSON object which the server can use to correctly build a container. In this example, the data contains the path to a directory the student begin in, i.e. their working directory.

This JSON object is then encrypted using a shared key and given to the student. This encrypted specification will be used to request an instance that fulfills that specification without exposing any information that should be kept secret.

The design to have an encrypted specification given to the user is intentional. Another choice would have been to have the creator of the specification directly provide it to the uAssign server. However, in the case of PrairieLearn, an unexpected server outage would mean that the specification generation is a failure while generating a question, which means a student will get a broken problem (possibly on an exam) and not be able to handle it. By having PrairieLearn generate the specification, it is always guaranteed to succeed, and potential outages of the uAssign server only impact the student (who can just wait for the server to come back up again). This is an important benefit for terminal problems in an exam, where an issue would be only temporary, rather than a permanent problem with the generated question (leading to manual intervention by the instructors).

Specification encoding and encryption is handled in PrairieLearn via a shared Python library. This allows PrairieLearn questions to generate specifications and treat them as any other string-based parameters given to the student. Listing 4.2 shows Python code that generates a question with the specification in Listing 4.1.

```
{
    "specID": "215b3567-f279-4699-97c9-c4978b8a6ae5",
    "assignmentName": "fs.pwd",
    "data": {
        "path": "foo/bar"
    }
}
```

Listing 4.1: An example assignment specification.

```python
import uassign

def generate(data):
    spec = uassign.create_spec('fs.pwd', {  # encode and encrypt
        'path': 'foo/bar',
    })

    data['params']['spec'] = spec
    data['correct_answers']['path'] = '/home/student/foo/bar'

    return data
```

Listing 4.2: Python code which generates the specification given in Listing 4.1 and defines the correct answer known in Listing 5.1.

Chapter 5 Assignment instances

From an assignment specification, the uAssign server creates an assignment *instance*. This instance represents the main state on which a student operates. An instance contains the ID of the container in use, as well as metadata that says whether or not the container exists on the host, if the container is running (i.e. the instance is active and a user is connected), as well as timing information that the server can use to know when to disconnect a user and clean up.

To create an instance, the server needs to perform a series of steps beginning with the provided specification and ending with a working container.

5.1 Instance API endpoint

In order to obtain an instance, the student needs to provide the specification to the server. To do this, a student uses an HTTP POST request, submitting the encrypted specification, aiming to receive an instance's UUID in return. The server, upon receiving this request, validates the encrypted payload (see Chapter 7.1), then processes the specification.

Once the server has received and validated the request, the server looks up the specification by UUID for existing instances. If one exists, then the server doesn't need to do any more work and returns the existing instance's UUID to the client. This existing instance may or may not be actively running and will be handled either way when the student connects to it (see Chapter 6).

5.2 Instance generation

If no instance exists, the uAssign server must create one. To do this, the server needs to convert the specification to a configuration that can be applied to a Docker container.

To facilitate this process, the server maintains a directory of assignment-related files. Each assignment has its own directory, and directories can be nested according to their name, where the dots indicate path separators (e.g. a name like `fs.pwd` would become `fs/pwd` on a host with `/` as its path separator, like Linux).

5.2.1 `index.js`

The uAssign server needs a way to convert incoming specifications into something it can use to produce a container. In early versions, uAssign did this by using templatized Dockerfiles, providing the specification's data as the template's context. But, this proved to be problematic for a number of reasons. Building Docker images using Docker's usual build process is very slow, and a Dockerfile template is hard to create correctly without eventually needing to deal with issues like syntax-significant whitespace and the lack of editor support for mixed Dockerfile/template engine code, making for a poor experience for both instructors and students.

Partway through the development process, the choice was made to change the way uAssign implemented its container configuration. The implementation needed to be flexible enough to handle more complicated assignments, while still being familiar to those developing them. The obvious choice was to use some sort of scripting language, which the server could run externally, or interpret itself. There are a number of different languages that could be used to varying degrees, including Lua, Python (or a derivative like Google's Skylark), or even a custom language, but in the end, uAssign uses JavaScript. A full explanation of uAssign's JavaScript runtime mechanism is given in Appendix A.

Each assignment directory is required to have a JavaScript source file named `index.js`. This file acts as a CommonJS module, exporting the functions required to generate a container configuration from a specification.

5.2.2 Generating a container configuration

As the first step of creating an instance, the uAssign server will call the module's exported `generate` function. This function has one argument, the arbitrary data object that is a part of the specification. This function returns a (JSON serializable) object describing how to construct the instance's container. The returned object is similar in function to Docker's own Dockerfiles, but is easier to create, offers more flexible actions, and gives better performance. Listing 5.1 shows an example of an `index.js` file which operates on a specification like Listing 4.1. Listing 5.2 shows a Dockerfile comparable to what Listing 5.1's `generate` function returns.

The returned object has the following properties:

`imageName` is a string containing the reference name of a Docker image. Docker images are typically hosted on Docker Hub, but can be referred to by fully qualified names (to pull from Quay, AWS ECR, etc). Providing an image name allows the server to simply "tag" the image with another name, skipping the slow Docker image building process. If an image name is not provided, the `dockerfile` property will be checked instead, but at least one must be provided.

`dockerfile` is a string containing a Dockerfile (not a path). This is given to the Docker daemon to build an image, similarly to using Docker normally. The Docker build process requires a build context (the files local to an image build), so the server will look for a directory named `context` next to `index.js` to provide to the Docker daemon. If this directory doesn't exist, then an empty context will be used. When providing the build context to the Docker daemon, `.dockerignore` is respected.

```
var _ = require("lodash");  // Import useful JavaScript libraries, if needed.

exports.generate = function(data) {
    var path = "/home/student/" + data.path;

    return {
        imageName: "jakebailey/ua-cs126-docker:latest",
        postBuild: [
            {
                action: "exec",
                user: "student",
                cmd: [ "mkdir", "-p", path ]
            }
        ],
        user: "student",
        cmd: [ "bash" ],
        env: [ "PS1=\\$ " ],  // Prevent the user from seeing their PWD at the prompt.
        workingDir: path
    };
};
```

Listing 5.1: An `index.js` which produces a container configuration from a specification like Listing 4.1.

`init` is a boolean that indicates whether or not the container should be given an internal init system. This is particularly useful in cases where the main program being run does not handle subprocesses correctly. If not provided, the default is true.

`postBuild` is an array of actions that will be performed in order on the container after its image is built, and it is running. In this state, the container has full internet access. The actions themselves are described in Chapter 5.3. If not provided, then no actions will be performed.

`cmd` is a string array which contains the main command a student interacts with. It is given in an "argv" form, similar to making an `exec` syscall in C-style languages. This must not be empty.

`user` is a string containing the username that should run the aforementioned command. It's not recommended to give access to the root user. If empty or not provided, then the user is inherited from the built image.

`env` is a string array which contains environment variables that should be present when running the command. Each element is of the form `KEY=VALUE`. The provided variables are merged with the environment variables that already exist in the image, overwriting if required. If empty or not provided, then the original environment variables will remain.

`workingDir` is a string containing the path of the working directory for the aforementioned command's execution. If empty or not provided, then the working directory is inherited from the built image.

```
FROM jakebailey/ua-cs126-docker
USER student
RUN mkdir -p /home/student/foo/bar
ENV PS1='\$ '
WORKDIR /home/student/foo/bar
CMD bash
```

Listing 5.2: A Dockerfile, equivalent to the object returned in Listing 5.1.

```
{
    action: "exec",
    user: "root",
    cmd: [ "some-command", "my_cool_config.conf" ],
    workingDir: "/etc/default",
    env: [ "FOO=BAR", "BAZ=QUX" ],
    stdin: " ... "
}
```

Listing 5.3: Example of an `exec` action.

5.3 Build actions

To configure containers, uAssign implements a number of "actions", which can be executed on a container to perform a specialized task. Each action is represented as a JavaScript object, where the `action` property identifies the name of the action being specified.

All actions make an assumption about the container they modify – that the system is POSIX-compliant in terms of its available programs. Most available Docker images include the needed tools (`sh`, `cat`, `tar`, and so on), but this should be taken into consideration while creating an assignment which makes use of these actions.

5.3.1 `exec` action

The `exec` action executes programs on the container. The command being run, the user running it, the environment, and the working directory are all configurable, and behave exactly as `cmd`, `user`, `env`, and `workingDir` in Chapter 5.2.

In addition to those options, the `exec` action also supports specifying the command's standard input via `stdin`, as a string. This can be used to give data to programs which expect it, without having to construct a complicated command to pipe the data in by some other means. If `stdin` isn't provided, then standard input won't be opened for the program.

Listing 5.3 shows an example of an `exec` action.

```
{
    action: "write",  // Or, append.
    user: "root",
    filename: "/path/to/file.bin",
    contents: btoa(someVariable),  // Encode some binary data.
    contentsBase64: true
}
```

Listing 5.4: Example of a `write` action.

5.3.2 `write`/`append` actions

The `write` and `append` actions write and append to files in the container, respectively. In these action objects, `filename` specifies the path to the file to be modified, and `contents` is a string that represents the contents that will be written/appended to the file. Optionally, the actions support base64-encoded file contents for cases where the data cannot be represented in a JavaScript string. Setting `contentsBase64` to `true` will tell the server to decode the contents before writing them.

Listing 5.4 shows an example of a `write` action.

5.3.3 `gobuild` action

The `gobuild` action builds Go source code into static binaries, then copies them to the container's binary directory. This is useful for adding a grading binary into the container which a student can run to check their work and produce a secret to prove the assignment is complete.

The server expects a directory called `gosrc` in the assignment directory, which contains Go source code.[1] The packages to be built are specified by `packages`, an array of fully-qualified Go package names (for example, `github.com/derekparker/delve/cmd/dlv`). The `gobuild` action also allows the Go compiler's "ldflags" to be specified [9]. This allows for programmatic modification of variables in the final Go binary (using the `-X` flag), without needing to templetize the source code itself. A common use case is to set secrets and parameters unique to the instance's grading binary.

Listing 5.5 shows an example of a `gobuild` action.

Go was chosen due to its fast compilation time and static binaries. As the uAssign server has access to a working Docker daemon, the server builds Go binaries inside a pre-created Docker image. This image accepts a Go source directory over standard input as a tar stream, compiles all specified packages, and outputs another tar stream containing all built binaries. These binaries are then passed to the instance's container through another tar call, extracting into `/bin`.

[1]In Go parlance, this directory is `$GOPATH/src`.

```
{
    action: "gobuild",
    packages: [ "helloworld" ],
    ldflags: "-X main.secret=hunter2"
}
```

Listing 5.5: Example of a `gobuild` action.

5.3.4 `parallel`/`ordered` actions

The `parallel` and `ordered` actions allow multiple actions to execute at once. Both require a property named `subactions`, which is an array of other actions, similar to the top-level `postBuild` array.

The `parallel` action takes the array of actions and executes them all at once. The action "completes" when all of the subactions have completed. If any action causes an error, then the other actions are canceled.

The `ordered` action takes the array of actions and executes them in order from first to last. The action is complete when all of the subactions have completed. If any action causes an error, then no more actions are executed.

The combination of these two actions allows for many actions to be executed at one time, while still preserving the ability to order actions that relate to each other. For example, a `gobuild` action can run and compile a program, while at the same time files and directories can be written that aren't directly related.

Listing 5.6 shows an example of both `parallel` and `ordered`.

5.4 Image build and container configuration

Once the uAssign server has executed the `generate` function in `index.js`, it can produce the Docker image and container to finish creating the instance. As mentioned in Chapter 5.2, either the object contains an image name, or a Dockerfile. If an image name was provided (most common), then the server will simply "tag" the image with a new name. This tag is an alias to the same image as the specified name. Deleting the tagged image will only delete the tag, allowing the same image to be shared between multiple instances without extra logic to manage them.

If a Dockerfile was provided, then the server will instead perform a "normal" Docker build. This build process uses the same process as the Docker CLI itself, the only difference being the location of the build context (its own directory, as previously mentioned). This process is slow, and not using a Dockerfile is strongly recommended (see Chapter 5.5).

Once the image tag or build is complete, the server creates a Docker container. Certain elements of a container's configuration can only be specified at creation, including resource limits (see Chapter 7.2) and initial network setup (which will be enabled during setup only).

Among those configuration options is the main command that the container will execute at startup. uAssign hardcodes this to be `/bin/cat`, such that the container will never exit (since `cat` will never receive an end-of-file marker without input). Any interaction done with the container is instead done by using an "exec" (normally exposed in the Docker command `docker exec`).

Once the container is created, the uAssign server starts the container, then executes all of the `postBuild` actions in order according to their individual specifications given in Chapter 5.3. When the actions complete without error, the uAssign server removes the `bridge` network from the container, disabling its internet access while preserving loopback.[2]

At this point, the container is ready for use. The server stops the container (leaving it on the host system) and stores the container's ID with the instance to be looked up later.

[2]This differs from disabling networking entirely, which can only be done at container creation time (not runtime) and completely removes all networking capabilities including loopback.

```
{
    action: "parallel",
    subactions: [
        {
            action: "gobuild",
            packages: [ "helloworld" ],
            ldflags: "-X main.secret=hunter2"
        },
        {
            action: "ordered",
            subactions: [
                {
                    action: "exec",
                    user: "root",
                    cmd: [ "mkdir", "-p", "/etc/some/path" ]
                },
                {
                    action: "write",
                    user: "root",
                    filename: "/etc/some/path/foobar.conf",
                    contents: "abcd = 12345\nxyz = 'test'"
                }
            ]
        }
    ]
}
```

Listing 5.6: Example of the `parallel` and `ordered` actions.

5.5 Differences in image building compared to Dockerfiles

Frequent users of Docker may notice that using Dockerfiles is not recommended. Instead, uAssign implements its own container configuration with its own actions. This custom implementation could be seen as a reimplementation of the wheel, but it comes with significant advantages.

Docker was designed with reuse in mind. Docker images are internally organized into layers, where each layer represents the difference between the last. When one layer changes, only those that follow it need to be changed. All of the previous layers can be reused, saving both CPU time, as well as network usage for those downloading the image from shared repositories. This matches Docker's most common use-cases, as an application deployment and management system where the same image may be used on thousands of machines and saving resources is important and as a development tool where the user experience is greatly improved by a reduced build time (like using `make` to reuse object files in a large C++ project).

However, uAssign cannot make use of this sort of efficiency. Most assignments do not share much state past their base containers (i.e. a shell, common utilities, perhaps a compiler). Each directive in a Dockerfile adds a layer, and each layer adds a non-trivial amount of time, since the Docker daemon needs to store that state for potential reuse.

By skipping this build process altogether and performing actions directly on the target container, uAssign reduces the build time significantly. And through the `parallel` action, many duties can be performed at once, while Dockerfiles are strictly ordered, executing one directive at a time.

Further, by tagging a base image with a new name and performing tasks directly on a container, no intermediary images are produced. When the image is no longer needed, it is deleted, which the Docker daemon implements by deleting the tag. In a normal Docker build, the intermediate parts are saved, even when the overall build fails. uAssign would need to track that in order to effectively manage resources, but not performing a normal build at all eliminates that need.

Chapter 6 User interface and terminal protocol

Although the uAssign server contains the bulk of the logic, the student's machine is tasked with performing the initial actions to access an assignment.

A potential client needs to perform the following steps:

1. The client sends an HTTP POST request to the server containing a specification (as in Chapter 5.1). On success, the server returns the UUID of an instance.

2. The client opens a WebSocket connection to the server, providing the instance UUID.

3. The client and server proxy input and output, until one side disconnects.

To implement these steps in PrairieLearn, uAssign includes a PrairieLearn element, which allows for easy embedding of HTML and JavaScript code which performs all of these steps.

6.1 PrairieLearn element

The PrairieLearn element is implemented like other elements and can be put into a course's elements directory. Listing 6.1 gives the HTML code required in a question's `question.html` to load the element, producing an interface like the one given previously in Figure 1.1.

The element supports a number of options:

`spec_path` is a string which contains the name of the param which contains the encoded specification that should be POSTed to the server (see Listing 4.2 for where that is defined). This is the only option that must be specified.

`host` is a string which gives the hostname of the uAssign server, for example, `term.example.edu`. This currently defaults to the production server, but can be changed to instead be `localhost` if needed for local testing. Note that this option should not be given a protocol (i.e. no `https`).

`insecure` is a string which contains either `true` or `false`, which is true if the host does not use HTTPS/WSS. The default is false, but can be overridden for testing where a certificate is difficult to obtain. For production use, the insecure mode should never be used.

```
<uassign_terminal spec_name="spec" />
```

Listing 6.1: The PrairieLearn HTML element that displays a terminal.

Under the hood, the element uses a library called `hterm`, a terminal emulator from the Chromium project written in pure JavaScript for use in the browser [6]. This is the same library which powers ChromeOS's SSH application and is well-tested.

The library itself does not know anything about the WebSocket or the uAssign server. An `hterm` terminal provides functions to write data to the terminal and hooks to capture input text and key combinations, as well as the terminal's dimensions. The PrairieLearn element hooks into these features, proxying data back and forth over a common protocol.

6.2 Terminal I/O protocol

The server and client communicate over a WebSocket, using a modified version of the protocol used by terminado [12]. Terminado is an application which serves a similar purpose to parts of the the PrairieLearn element, proxing a terminal between a server (though in its case, a Python Tornado server) and a client.[3]

WebSockets work in terms of discrete messages. In practice, it's difficult to use WebSockets without requiring that the messages be text strings only, and using JavaScript at one end requires that all strings be a valid Unicode encoding.[4] This limits the choice in protocol to be text, which terminado's satisfies.

Each message is JSON, encoded and decoded over the wire. Specifically, each message is a JSON array, where the first element in the array is a string defining the message's purpose and where further elements are extra information (like arguments). The modified protocol supports four message types:

`stdin` is sent from the client to the server to provide standard input. It takes one extra array element, a string containing the input. A message of this type may look like: `["stdin", "pwd\n"]`

`stdout` is sent from the server to the client to provide output to the terminal. It takes one extra array element, a string containing the output. Although the name would suggest that only standard output is sent over this message, the server will interleave standard output and standard error (as a real terminal would) and display it as such. A message of this type may look like: `["stdout", "/home/student/foo/bar\n"]`

`set_size` is sent from the client to the server to set the dimensions of the emulated TTY. It takes two extra array elements, a height and a width, both positive integers. This message is typically sent at the beginning of a connection, as well as periodically as the viewport changes. A message of this type may look like: `["set_size", 24, 80]`

[3]This protocol was chosen at an early stage of development of uAssign, as a similar JavaScript terminal library was used until switching to `hterm`. Another protocol could be used instead, but terminado's is efficient enough for uAssign's use and can be extended to serve other purporses.

[4]This fact leads to complications when the server isn't also implemented in JavaScript. In our case, uAssign is implemented in Go, which requires all strings to be UTF-8. This poses a problem when sending data to the client as multi-character symbols could be sent in pieces, causing the receiving JavaScript to fail to convert the data to its own UTF-16 representation. This problem is solved server-side by only allowing fully completed Unicode symbols to be sent.

`wipe` is sent from the server to the client, instructing the client to clear its terminal screen. This functionality is already built into the terminal emulator, but this extra message allows the uAssign server to wipe the terminal without needing to send a cryptic set of control codes via `stdout`. In practice, this is used to show a temporary "please wait" message while the instance is brought online. A message of this type may look like: `["wipe"]`

6.3 Additional features

The PrairieLearn element also supports a few extra features. Most notably, the terminal includes a "reset" button. When pressed, the client will fire off a request to the server, instructing it to force an early cleanup of the active instance. Once the instance has been removed, the client will repeat its initial POST/WebSocket requests. Since the instance was removed, the student will be presented with another instance of the same specification, effectively resetting the assignment.

Additionally, the element will capture errors that the server produces and display them to the student. The uAssign server adds a request ID header to all requests; the student can be given the ID to aid in debugging without revealing internal details.

Chapter 7 Security concerns and misuse protection

uAssign was designed to be as secure as possible while still providing students with an accurate environment to complete their assignments. Since uAssign gives a user access to a real container on a real machine, special care must be taken to ensure that that privilege is not abused.

uAssign, much like every other service which operates on the open internet, needs to protect against general third-party attacks. As such, it implements standard security measures, including HTTPS (with built-in LetsEncrypt support for free certificate issuance), Secure WebSockets, and a CORS policy, all without needing to set up a complicated HTTP proxy like Apache or NGINX. But, in addition to those third-party attacks, uAssign also needs to be able to protect against potential attacks by its users on the assignments themselves, as well as the resources that the assignments use.

7.1 Protecting specifications

One crucial element that uAssign needs to protect is the confidentiality of the assignment details themselves, namely the assignment specification. The specification contains all of the information needed to "solve" the problem, so a student that's able to read it will be able to skip doing the assignment entirely and simply answer with whatever secret the specification includes. uAssign is designed in such a way that it requires something like PrairieLearn to give a student the specification to later send to the uAssign server, and since the user cannot be trusted, the specification needs to be encrypted.

To protect the specification, uAssign (and its libraries) include support for encryption through a custom library called `simplecrypto`. This library has implementations for the main languages used in uAssign and PrairieLearn (Go, JavaScript, and Python), and was created to thinly wrap around each language's built-in encryption features to keep the implementation auditable and not an instance of "roll-your-own crypto". This library only supports shared-key encryption via AES in CFB mode with HMAC-SHA-256 for verification. It also defines a serialized format in JSON, which contains the ciphertext and its HMAC for verification.

When deploying a uAssign server and its associated PrairieLearn libraries, an AES key is generated and given to both. When PrairieLearn creates a specification, it encrypts and serializes it before giving it to a student. When the student submits the specification, the encrypted payload is what they observe, keeping the specification's details private and preventing forgery.

Additionally, during the development of the course material for uAssign, we found that some students were able to extract strings from our Go grading binaries, giving them a very easy way to obtain secrets. In order to mitigate this sort of attack, we added obfuscation support to uAssign's JavaScript runtime through an extra built-in function called `gzipXorBase64`. This function obfuscates a string by gzipping it, XORing it against some mask, then base64 encoding the resulting bytes back into a string (see Appendix A). This string is then compiled into the grading binary which is able to undo the process. A method like this is not foolproof, however the amount of effort required to bypass doing the assignment would be significantly more difficult than just doing the assignment itself.

7.2 Container resource limits

uAssign must also make efforts to prevent the containers themselves from becoming a risk. It does this by using Docker's own resource limiting features [4]. The Docker API provides support for:

- Memory usage limits, by specifying a hard maximum in bytes.
- CPU quotas, by specifying what percentage of the system a container is allowed to use before being unscheduled.
- Process limits (similar to `ulimit`), specifying the maximum number of processes a container can spawn.

These options cover the bulk of potential problems, however, a container also has access to disk storage. Docker has the ability to limit a container's storage usage, but only at the storage driver level.[5] Unfortunately, this setting is not per-container and must be applied to Docker as a whole when deploying a uAssign host system.

Docker also manages networking for its containers, creating networks and virtual network adapters. In the case of uAssign, networking is available to a container while it is being configured, but is disabled before a student has access (as mentioned in Chapter 5.4).

7.3 Instance mutual exclusion

In addition to limits placed on containers themselves, uAssign adds an extra limitation to instances. An instance is only allowed to be connected to a single terminal at a time. If another client attempts to connect to the same instance, the first connection is terminated, and the second is takes its place. This prevents uAssign from being used for unintended purposes that involve multiple connections to the same container, such as a makeshift chat room.

[5]In Docker, a storage driver is something which implements a common storage API to give the daemon access to volumes. This includes a driver for `device-mapper`, `btrfs`, `OverlayFS`, and others.

Chapter 8 Results and performance characterization

This section covers various datasets and testing involving uAssign and its current use, including a student survey and various performance tests.

8.1 Student survey

During the Spring 2018 semester, a survey was conducted of CS 126 students who had used uAssign in homework assignments. This survey covered personal preferences (operating system and editor choice), as well as confidence in terminal skills and uAssign's usability. The complete set of questions can be found in Appendix B and its IRB approval notice in Appendix C. In total, 57 students participated out of an enrollment of 264, constituting a respondent rate of 22%.

One set of questions asked students to self-rate their proficiency in the terminal on a scale from 1 to 5 (from "no experience" to "very experienced"), both before and after completing uAssign's terminal assignments. This data is shown in Figures 8.1 and 8.2.

By taking the difference between these two datapoints, we can get an idea of how much students feel they have improved by completing the assignments. Overall, on a scale from 1 to 5, the average change in confidence reported by students was just over a point in the positive direction, at 1.02. When combining this metric with the students operating demographics information (Table 8.1), Windows users self-report benefiting the most from the terminal assignments, followed by MacOS users. Users who used Linux as their primary operating system reported no improvement, likely because they were already proficient.

Another interesting result is the students' opinion of terminal skills in general, shown in Figure 8.4. No student surveyed rated the value of terminal skills below a 3. Overall, 95% of participants rated the value of terminal skills at a 4 or higher, with 72% of the participants giving a rating of 5. This is a significant result,

Primary		Primary + Secondary	
OS (% of participants)	Avg. Improvement	Secondary OS	Avg. Improvement
Windows (54.39%)	+1.323	Linux (15.79%)	+1.889
		N/A (36.84%)	+1.095
		MacOS (1.75%)	+1.0
MacOS (40.35%)	+0.739	N/A (28.07%)	+0.875
		Windows (3.51%)	+0.5
		Linux (8.77%)	+0.4
Linux (5.26%)	+0.0	Windows (5.26%)	+0.0

Table 8.1: Self-reported improvement in terminal skill confidence, by operating system.

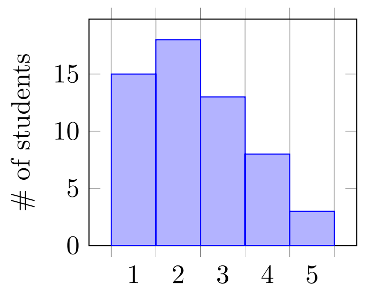


Figure 8.1: "Rate your proficiency with the command line before CS 126."

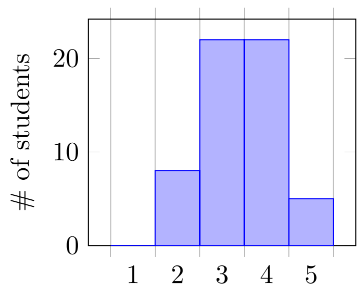


Figure 8.2: "Rate your proficiency with the command line after the terminal assignments."

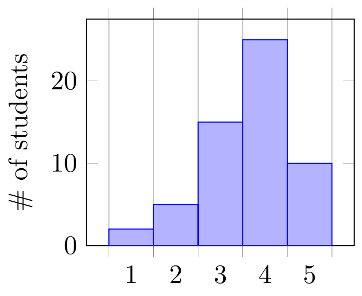


Figure 8.3: "My time was well-spent completing terminal assignments."

showing that the vast majority of students see the terminal as a useful tool.

Questions about uAssign and its PrairieLearn integration specifically are also telling. Almost 90% of participants said that the terminal interface was user friendly (Figure 8.8). 77% answered that the interface was bug-free, with 7% disagreeing (Figure 8.9). Crucially, almost 90% of students answered that they would prefer for their terminal assignments to be delivered in PrairieLearn as opposed to another homework platform, with only one participant disagreeing (Figure 8.10).

Demographic questions about student workflows have varied responses. Most students said they preferred GUI editors over terminal editors and preferred doing basic tasks with a GUI rather than using a terminal. 61% of students appear to not have a preferred terminal editor at all. Those with a preferred GUI editor mostly preferred using an IDE, with the rest split between Atom, VS Code, and Notepad++. This makes sense for a number of reasons. Most students reported that they had little to no experience using the terminal, and CS 126 is largely a freshman-level course following CS 125 (Illinois' introductory CS course) which teaches Java and promotes IDEs like IntelliJ and Eclipse. CS 126 continues using Java, but transitions to C++ late in the semester.

Other question responses show room for improvement. While 60% of the participants stated that they felt confident they could use common terminal utilities, 40% still either disagree or are unsure (Figure 8.5). 51% were confident that they knew the right terminology to ask questions if they had issues in the terminal, but the remaining 49% is a cause for concern (Figure 8.7).

8.2 In-class performance testing

uAssign has been successfully used for the past two semesters; Fall 2017 and Spring 2018. A number of performance problems were discovered and addressed in the first semester, and further improvements to the assignments themselves occurred in the following semester. In Spring 2018, we performed an in-lecture stress test, where students in both lectures were asked to access their terminal assignments all at once and attempt

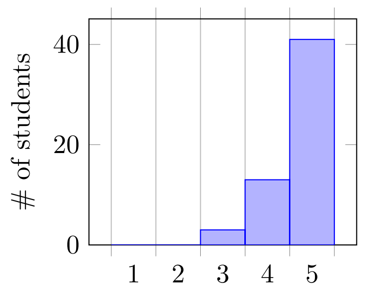


Figure 8.4: "I believe terminal skills are useful."

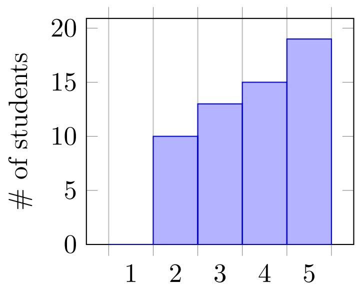


Figure 8.5: "I am confident that I can use common terminal utilities."

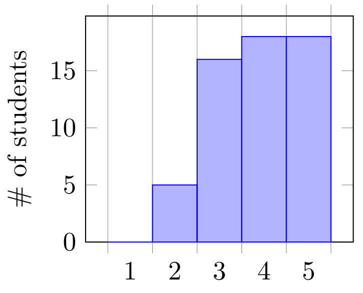


Figure 8.6: "I am confident that I can read terminal utility 'help' documentation (`-h`, `-help`) and man pages."

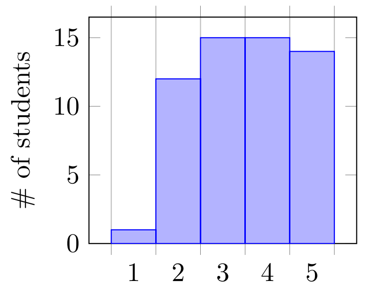


Figure 8.7: "I am confident I will know the right terminology to ask questions about issues I have in the terminal."

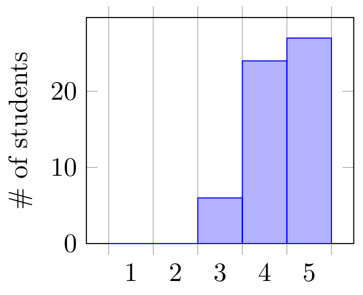


Figure 8.8: "The terminal assignment system provided a friendly user interface."

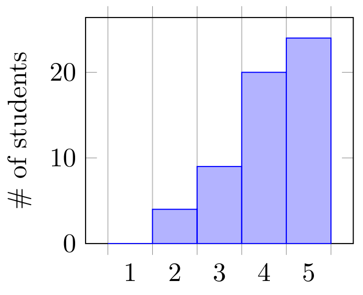


Figure 8.9: "The terminal interface provided on PrairieLearn was bug free."

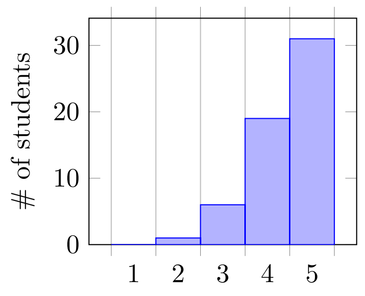


Figure 8.10: "I prefer to have terminal assignments in PrairieLearn as opposed a different website/homework system."

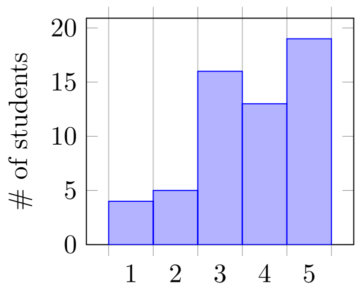


Figure 8.11: "I prefer dedicated editors/IDEs such as Atom and IntelliJ over terminal-based editors like vim and emacs."

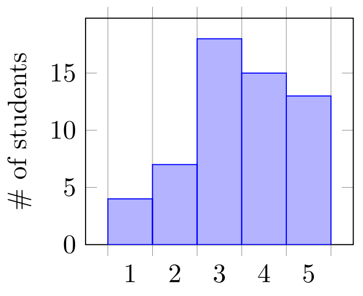


Figure 8.12: "I prefer performing common tasks (like copying/opening/deleting files) in a GUI, rather than in the terminal."

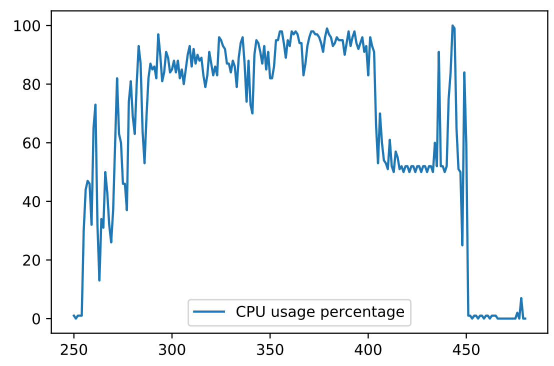


Figure 8.13: The first lecture test's CPU usage.

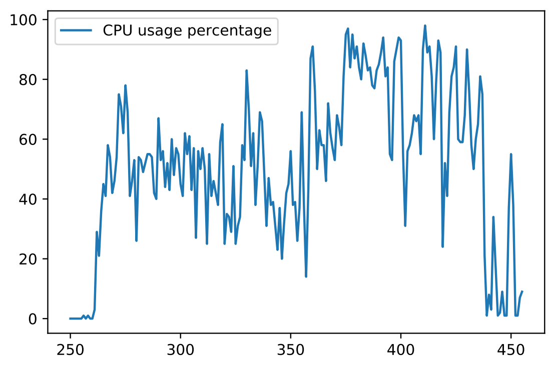


Figure 8.14: The second lecture test's CPU usage.

to overload the server. Each lecture involved roughly 80 concurrent users completing assignments. In both tests, uAssign performed well on its AWS `t2.medium` instance, with assignments building in a few seconds and staying responsive once built.[6]

To collect data during this test, the uAssign host ran an instance of `collectl` [22], a lightweight monitoring tool which collects a wide range of data, including but not limited to CPU usage, memory statistics, and network usage. Figures 8.13, 8.15, and 8.17 show metrics from the first lecture's test, and Figures 8.14, 8.16, and 8.18 show metrics from the second lecture's test.

This sort of load is unusual, as students typically complete assignments on their own schedule over the course of many days. Having a significant number of students create new assignment instances all at once strains the CPU and disk of the host machine, as it needs to spin up new containers, which in our current deployment means creating virtual volumes with their own filesystems. Further deployment improvements could be made to limit this by instead making use of a filesystem with native Docker storage support like `btrfs`, where no extra work is needed to create a container's storage space. The memory usage of the server only increased a few hundred megabytes, and network usage was relatively low, meaning that future deployments need to prioritize computing power and I/O over other resources.

[6]The `t2.medium` AWS instance type only has 2 virtual CPUs and 4GB of memory. A larger instance would likely perform even better.

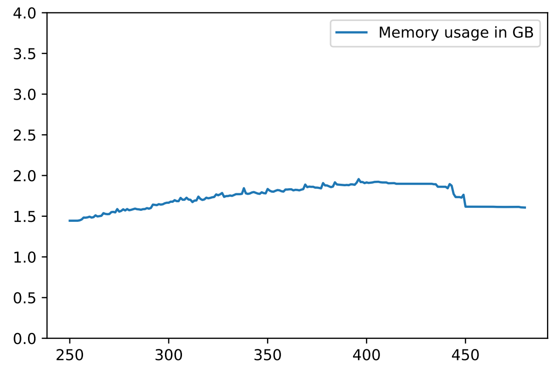


Figure 8.15: The first lecture test's memory usage.

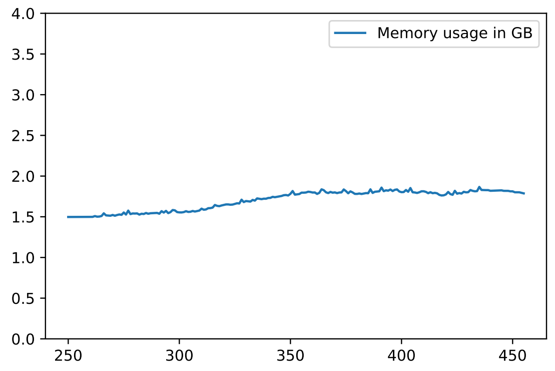


Figure 8.16: The second lecture test's memory usage.

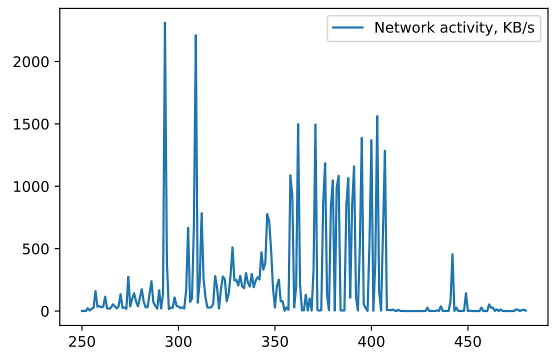


Figure 8.17: The first lecture test's network usage.

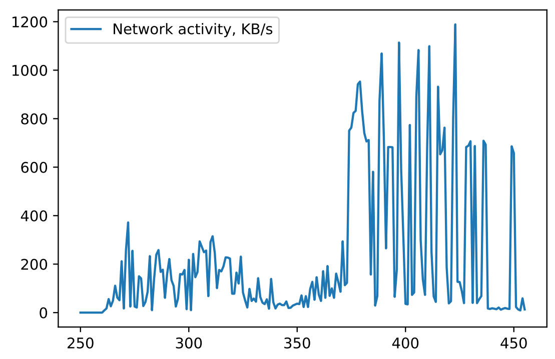


Figure 8.18: The second lecture test's network usage.

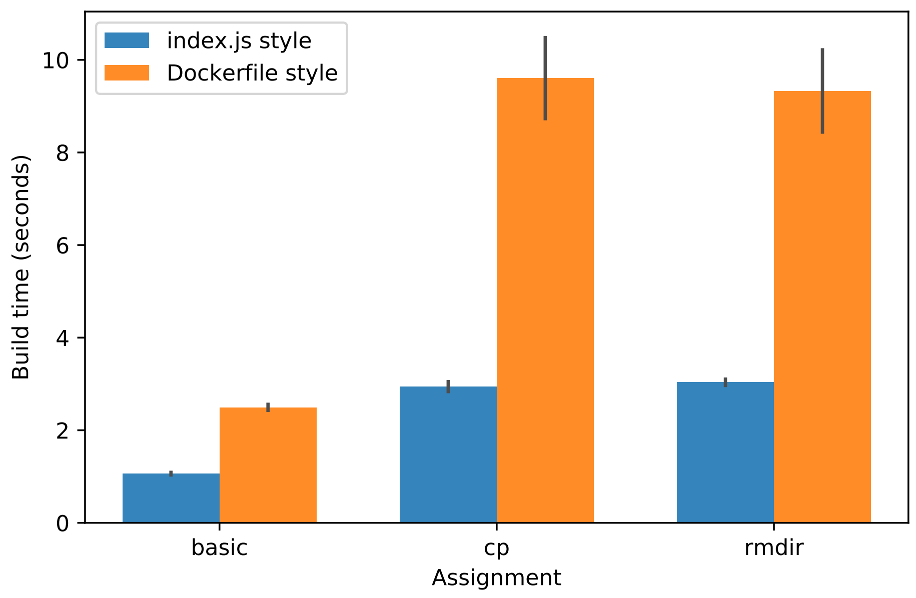


Figure 8.19: A comparison of build times using uAssign's actions and Docker's own build system.

8.3 Actions compared to Dockerfile build times

uAssign's choice to use its own actions to configure containers as opposed to Docker's own build process gives it a significant speed advantage and therefore greater usability for students. To show the performance difference, three assignments were backported from an `index.js` style to a style that uses Dockerfiles instead:

- The "basic" assignment asks students to read a file and answer with its contents.
- The "cp" assignment asks students to copy a file from one location to another, thereby renaming it. It involves building a Go binary for grading.
- The "rmdir" assignment asks students to remove a directory. Similarly to the previous assignment, it needs a Go binary to check that the specified directory was deleted.

The results are shown in Figure 8.19. Overall, uAssign's implementation results in a 2 to 3 times speed up relative to a traditional Docker build.

Chapter 9 Related work

Since the creation and adoption of UNIX and its derivative platforms like Linux/BSD/MacOS, many methods to teach and assess terminal skills have been developed and used. These methods began with more traditional courses and have only within the last decade expanded into complete web-based systems.

9.1 Early examples

One of the earliest examples of a platform to teach UNIX skills was *learn* [15], created by Brian Kernighan (of K&R C and awk fame) in 1979. Included with UNIX v7, *learn* created an interactive environment on a user's system aimed at teaching the usage of the system it ran on by interpreting "CAI" (computer aided instruction) scripts. Lessons varied from basic command usage and filesystem navigation to the C programming language. This software was intended for self-guided instruction only and lacked any sort of assessment outside of completion of a given lesson, nor appeared to be a part of any coursework.

Many later examples teach terminal skills using more traditional teaching methods, using in-class lessons with hand-graded quizzes [14] [20]. These methods may provide value, but are limited to what a given course staff can manage. Additionally, using quizzes (be they paper or online) may not fully assess skills, relying on a student to answer a question by statement, rather than perform some task.

9.2 Moving into the cloud

As computing education becomes more popular, scaling the teaching and assessment also becomes a problem that needs to be solved. A common technique is to move assignments into "the cloud", providing environments for students to access.

One area that often develops tools to teach the terminal is IT and systems administration. In this subject, it's often a main goal to provide access to expendable systems, such that learners don't break actual systems. To this end, methods used are often based on VMs.

In 2004, "User Mode Linux" (U-ML) was developed at the University of Bologna Italy to teach OS administration [11]. This custom Linux derivative exists as a virtual machine. Students connect to these virtual machines over emulated serial lines provided by the VM host, where students can perform administration tasks such as simulated system updates, service management, and more. While it implements some fairly modern features (a copy-on-write filesystem to share assignment materials between students, control of networking to allow or disallow internet access), the implementation is fully custom, complicated, and outdated. Additionally, the assessment methods used to determine if a given assignment was completed correctly is unspecified.

A more recent example from IIIT Hyderabad in 2011 approaches a similar IT teaching problem using virtual machines [10]. Unlike U-ML, IIIT Hyderabad's "Virtual Lab" uses a web interface to provide access to its virtualized environments. The client browser uses AJAX requests to send a user's specified command to the server. The server then uses SSH to run the command, captures its full output, then returns the output back to the user as an HTTP response. However, this interface is not a "true" terminal, as it is not an interactive session (i.e. you cannot run editors, live-updating programs like `top`, nor cancel execution or see live output of long-running commands). Although this provides an advantage over Bologna's U-ML, as it uses the browser as opposed to a serial emulator, it limits what that a user can do, and is a departure from a "real" terminal that a student may use in other contexts. Assessment is provided in the form of online quizzes in the same user interface as the emulated terminal, however their format and contents is unspecified.

9.3 Parameterized assignments

Although the previously mentioned methods contain elements which are important to teach the terminal (isolation, real access, scalability), they don't parameterize their assignments or provide instant feedback/autograding, instead relying on pre-created assignments and quizzes.

A different approach is taken by "unix-training", from Matthieu Moy at the Grenoble Institute of Technology in 2011 [19]. This method presents its lessons as a treasure hunt, where each step gives you information that can help you find the next solution. These steps include finding resources on the internet, compiling code, editing broken source code, as well as creating shell pipelines and accessing remote systems over SSH. The steps each student needs to take are unique to them, generated psuedorandomly (using some hashing of user information), which prevents copy-paste cheating. To assess students, students enter solutions into a webapp which records their progress as the work through their challenge and provides instant feedback about their answers. Additionally, a different assignment is generated for an exam, which covers the same material as the lessons. Parameterizing and providing instant feedback is a step in the right direction, however this system relies on students having their own *NIX machines for course work, and, in the case of exams, relies on populating each machine's filesystem with a particular student's exam.[7]

9.4 TuxLab

TuxLab, a student project at Carnegie Mellon University (CMU), is a complete interactive "Linux course" framework [1]. TuxLab is the most analogous system in use compared to uAssign, and was developed

[7]Pre-populating exam machines with student exams was also a solution used in early CBTF exams at Illinois, but was replaced with PrairieLearn soon after.

concurrently.

TuxLab provides a complete online learning and assignment system. It is entirely self-contained, providing its own user interface for lessons, a combination of Docker and SSH to provide a running system a user can access, and supports autograding of container state.

At a high level, TuxLab works by

- running instructor-defined JavaScript (referred to as "Labfiles") that configures a Docker container for an assignment, and
- starting an SSH server on the Docker container and connecting client-side to that server with a JavaScript SSH client via a specified IP/username/password combination, and
- periodically grading the container's state using JavaScript grading functions defined in the "Labfile" for the given assignment, which provides instant feedback as well as a way to assess/grade the student's work.

9.4.1 In-depth comparison

Although TuxLab and uAssign are similar in many ways, there are some distinct differences between the two in overall design and underlying implementation.

Beginning with similarities, both uAssign and TuxLab work off of the same core principle – exposing customized Docker containers to students to complete terminal assignments. Both use JavaScript code written by instructors to configure containers (though in different ways), and both configure containers by directly performing tasks on the container before the student connects, thereby skipping the lengthy Docker build process. Both set limitations on the containers in terms of resource uses to prevent any one student from affecting another and impose some amount of abuse prevention (both set CPU/memory limits, but uAssign's current deployment also limits disk usage).

TuxLab itself is a complete learning webapp, complete with lessons, assignments, and grades. uAssign, on the other hand, is a component that is meant to be used through existing assignment systems (mainly, PrairieLearn). This gives TuxLab an extra level of control and a more interesting user interface (a terminal on the right, reference on the left), while uAssign benefits from pairing with the same platform as our students' other assignments (which was found to be a positive design decision during the survey covered in Chapter 8.1) and does not need to re-implement a full front-end.

In addition to managing a container's configuration, TuxLab's Labfiles also provide the ability to grade the container state for correctness directly, while uAssign does not. uAssign instead relies on binaries within the container to determine if the current state is correct, adding some extra build time to produce that binary.[8] The added grade function is a good feature, as it allows grading to be quick (no extra binary) and away from potential student meddling. uAssign omits this feature due to the design choice to make it plug into PrairieLearn. Since PrairieLearn is what determines the final grade, it would need to contact the uAssign server to ask it if the container is correct. PriaireLearn typically can't "fail" to grade one of its own questions, so a temporary outage of the uAssign server would cause PrairieLearn to enter a bad state. TuxLab avoids this problem altogether by doing everything itself, including both the assignment hosting and grading.

The way TuxLab gives student access to its containers is significantly different than uAssign's. TuxLab's Docker containers run in Docker Swarm. To connect to a container, a student connects via SSH (with an in-browser SSH client) to an SSH proxy on the TuxLab server, which then connects to the container. This is required due to the way Swarm operates, abstracting containers to "services", where the only interaction with a container's I/O is through Docker-collected logs or though the internet to a server running in that service, at an IP address in the Swarm network that is essentially randomized. In terms of deployment and maintenance, this is a simple design, but in effect means that the containers have full network access. At first glance, the inclusion of an SSH proxy seems to hint that the containers do not have internet access, but inspection of documentation and code reveals that in the current implementation, TuxLab containers continue to have internet access even after setup. This opens up assignment instances to unwanted capabilities such as package downloads (usurping any instructor-defined setup), communication with outside help, or even malicious servers.[9]

Barring the security implications of TuxLab's setup, running an SSH server on each container does come with a compelling benefit. In addition to the SSH client, TuxLab lists the IP/username/password combination for their container. This means that students can use their own terminal emulators, which is likely a better user experience than an in-browser version. uAssign could implement a feature like this and run a psuedo-SSH server on an alternate port, at the expense of some added complexity.[10]

uAssign comes with a few more benefits. Instances in uAssign have well-defined statefulness. If a student leaves the question page for their terminal assignment and then returns, they will be presented with the same container again, with all of their work still saved. Only after a long period of inactivity will uAssign decide to clean up the resources. TuxLab does not do this, largely due to its use of Docker Swarm, where services are intended to be stateless and be spun up and down at any time (good for deploying your next big project at scale, but not for this specific purpose).

In uAssign, JavaScript code used for configuring containers is run in a sandbox, inside of the server itself

[8]And, this is done at the assignment implementation level, done by the instructor, which isn't ideal.

[9]Not even port blocking will completely solve these issues, since an attacker could simply use a reverse proxy to push traffic past the firewall over an outgoing connection and use the container's compute time for their own nefarious purposes.

[10]This is easier than it sounds, since there are standardized libraries for building SSH servers in Go.

(see Appendix A). TuxLab itself is written in JavaScript and chooses to run the Labfiles directly along side the server code. This could create a scenario where an instructor unwittingly writes broken code (for example, an infinite loop), adversely affecting the rest of the server. uAssign's sandboxing allows it to easily kill any misbehaving JavaScript. Even further, the JavaScript runtime uAssign uses is itself written in Go, which means that the Go runtime can schedule any other work it needs to do, kicking the misbehaving code out.

Finally, uAssign takes care to manage its Docker resources, assuming it will run indefinitely. It will periodically prune any unneeded containers, networks, and volumes and allows for preemptive cleanup of instances when a client knows they are no longer needed (for example, PrairieLearn can send a cleanup request when it sees that a student has graded their question and cannot revisit it). uAssign can also keep images up-to-date by autopulling recently used images, as the `generate` function in an assignment's `index.js` specifies the image name.

Chapter 10 Conclusion

Through the development of uAssign, it's been shown that it's possible to develop a system to effectively allow students to practice and assess their terminal skills in a way that is both performant and secure. uAssign gives instructors the opportunity to add terminal-based content to their own PrairieLearn material, without sacrificing flexibility in implementation. Through surveys, it's clear that there is a benefit for students, especially those who have not been exposed to the command line, and that the current workflow is positive and worth continuing.

10.1 Future work

uAssign is in its early stages. There are many different possible enhancements that can be made to the system, and topics to further study.

The low hanging fruit include supporting multiple courses and expanding past CS 126. Currently, the uAssign server assumes a single course, with a single server-side assignment directory. Modifications could be made to the uAssign server to use different encryption keys and assignment directories for each course. Terminal assignments could be integrated into future CBTF exams by whitelisting the current uAssign server. Or, uAssign could me merged into PrairieLearn's own featureset, opening it for use in any class that wants it.

Currently, existing terminal assignments only cover basic terminal usage. Since uAssign's containers are configurable, assignments could be expanded to cover more interesting topics. Containers keep their loopback network adapter, so an HTTP or git server could be started alongside a shell to assess skills which would normally require internet access. The protocol that uAssign uses to communicate with a terminal emulator could be extended to include more data and make it possible for an editor on a PrairieLearn question page to modify files in the container.

uAssign proxies all input and output from its containers to students. Because of this design, it is in the unique position to observe this interaction. Sessions could be recorded and saved for future study and curriculum improvements. This is somewhat similar to a common tool known as `script`, which can record a terminal session locally, with the self-proclaimed purpose of being "useful for students who need a hardcopy record of an interactive session as proof of an assignment" [7].

Past implementation details, there is much research to be done on the content of terminal assignments, and their effect on students as they move through their computer science curriculum. Now that a working assignment system exists, this sort of research is possible.

References

[1] TuxLab. http://tuxlab.org. Accessed: 2018-03-18.

[2] Computer-based testing facility (CBTF). https://cbtf.engr.illinois.edu/. Accessed: 2018-04-01.

[3] Docker. https://www.docker.com/, . Accessed: 2018-04-02.

[4] Limit a container's resources | docker documentation. https://docs.docker.com/config/containers/resource_constraints/, . Accessed: 2018-04-02.

[5] Dockerfile reference. https://docs.docker.com/engine/reference/builder/, . Accessed: 2018-04-01.

[6] hterm. https://chromium.googlesource.com/apps/libapps/+/master/hterm. Accessed: 2018-04-01.

[7] script(1) - linux manual page. http://man7.org/linux/man-pages/man1/script.1.html. Accessed: 2018-04-02.

[8] Ieee standard for information technology–portable operating system interface (posix(r)) base specifications, issue 7. *IEEE Std 1003.1-2017 (Revision of IEEE Std 1003.1-2008)*, pages 1–3951, Jan 2018. doi: 10.1109/IEEESTD.2018.8277153.

[9] The Go Authors. link - The Go Programming Language. https://golang.org/cmd/link/. Accessed: 2018-03-31.

[10] K. C. Bandi, A. K. Nori, V. Choppella, and S. Kode. A virtual laboratory for teaching linux on the web. In *2011 IEEE International Conference on Technology for Education*, pages 212–215, July 2011. doi: 10.1109/T4E.2011.41.

[11] Renzo Davoli. Teaching operating systems administration with user mode linux. In *Proceedings of the 9th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, ITiCSE '04, pages 112–116, New York, NY, USA, 2004. ACM. ISBN 1-58113-836-9. doi: 10.1145/1007996.1008027. URL http://doi.acm.org/10.1145/1007996.1008027.

[12] Jupyter development team and Ramalingam Saravanan. terminado. https://github.com/jupyter/terminado. Accessed: 2018-04-01.

[13] I. Fette and A. Melnikov. The websocket protocol, December 2011. URL http://tools.ietf.org/rfc/rfc6455.txt. RFC6455.

[14] Tyson Kendon and Ben Stephenson. Unix literacy for first-year computer science students. In *Proceedings of the 21st Western Canadian Conference on Computing Education*, WCCCE '16, pages 14:1–14:4, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4355-8. doi: 10.1145/2910925.2910930. URL `http://doi.acm.org/10.1145/2910925.2910930`.

[15] Brian W. Kernighan and Michael E. Lesk. Learn — computer-aided instruction on UNIX. In *UNIX Programmer's Manual*, volume 2. 1979. URL `https://s3.amazonaws.com/plan9-bell-labs/7thEdMan/v7vol2a.pdf`.

[16] Andreas Klöckner. RELATE. `https://documen.tician.de/relate/`. Accessed: 2018-04-06.

[17] P. Leach, M. Mealling, and R. Salz. A universally unique identifier (uuid) urn namespace, July 2005. URL `http://tools.ietf.org/rfc/rfc4122.txt`. RFC4122.

[18] M. D. McIlroy, E. N. Pinson, and B. A. Tague. Unix time-sharing system forward. *The Bell System Technical Journal*, 57(6, part 2):p.1902, 1978.

[19] Matthieu Moy. Efficient and playful tools to teach unix to new students. In *Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education*, ITiCSE '11, pages 93–97, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0697-3. doi: 10.1145/1999747.1999776. URL `http://doi.acm.org/10.1145/1999747.1999776`. Source code/wiki at `http://matthieu-moy.fr/spip/?Unix-training-a-set-of-tools-to`.

[20] Lawrence Osborne. Teaching c with unix for college credit to professional programmers. *SIGCSE Bull.*, 24(4):43–48, December 1992. ISSN 0097-8418. doi: 10.1145/141837.141852. URL `http://doi.acm.org/10.1145/141837.141852`.

[21] Dmitry Panov and Robert Krimen. goja. `https://github.com/dop251/goja`. Accessed: 2018-04-02.

[22] Mark Seger. collectl. `http://collectl.sourceforge.net/`. Accessed: 2018-04-02.

[23] Matthew West, Geoffrey L. Herman, and Craig Zilles. Prairielearn: Mastery-based online problem solving with adaptive scoring and recommendations driven by machine learning. In *2015 ASEE Annual Conference & Exposition*, number 10.18260/p.24575, Seattle, Washington, June 2015. ASEE Conferences. https://peer.asee.org/24575.

[24] Craig Zilles, Matthew West, and David Mussulman. Student behavior in selecting an exam time in a computer-based testing facility. In *2016 ASEE Annual Conference & Exposition*, number 10.18260/p.25896, New Orleans, Louisiana, June 2016. ASEE Conferences. https://peer.asee.org/25896.

Appendix A JavaScript runtime

uAssign uses JavaScript, a widely used language with a plethora of projects that allow it to be used as an embeddable language. The server uses a Go library called `goja` [21], which implements ECMAScript 5.1 in native Go, without any external calls to something like a C library or an outside NodeJS binary. Since `goja` is a Go library, using it adds no external dependencies to the final server, allowing for simple deployment.

`goja` supports a number of features which make it useful. It allows running code to be interrupted asynchronously, letting uAssign cancel a running build early without wasting time, or stop the code when a timeout is reached, protecting against broken assignment code. It implements things like exception handling (exposed to Go), access to randomness, `JSON.stringify`, as well as calls from JavaScript into Go and back again.

By the same authors, the `goja_nodejs` extension adds more features to `goja`, including support for CommonJS modules and the associated `require` function. This allows for some level of code reuse in assignments, as well as the potential to call commonly used JavaScript libraries like `lodash`.[11]

uAssign makes uses of these features, but also adds its own. `goja_nodejs` adds console logging support to the runtime, but only logs directly to the program's standard output. This is undesirable, especially since uAssign uses leveled logging where all output is JSON to be collected for analysis or debugging. uAssign instead includes its own version of the console library, which allows for logging to be caught per-runtime instance and therefore be exposed if an error occurs during the execution of `index.js`, logged for later retrieval with other errors.

uAssign also inserts the `atob` and `btoa` functions to handle base64 encoding, similar to a browser. It also adds a special function named `gzipXorBase64`, which accepts a string and an 8-bit number, gzips the string, XORs the compressed data (for obfuscation), then base64 encodes the data back into another string to be returned. All of these functions are implemented natively in Go using Go's standard library, and are called directly by the `goja` runtime making them faster than a pure-JavaScript implementation.

[11]uAssign includes `lodash` in its binary as an embedded resource, but others can be placed along side `index.js` in the assignment directory if needed.

Appendix B Student survey questions

The student survey covered in Chapter 8.1 consisted of the following questions:

B.1 Multiple choice questions

- What do you consider to be your “main” operating system?
  - ☐ Windows
  - ☐ MacOS / OS X
  - ☐ Linux
  - ☐ Other (please specify)
- If you use more than one operating system, what do you consider to be your “secondary” operating system?
  - ☐ I do not use more than one operating system.
  - ☐ Windows
  - ☐ MacOS / OS X
  - ☐ Linux
  - ☐ Other (please specify)
- What is your preferred GUI-based editor, if any?
  - ☐ I do not use a GUI editor.
  - ☐ Atom
  - ☐ VS Code
  - ☐ Sublime
  - ☐ gedit
  - ☐ Notepad++
  - ☐ Some IDE (Eclipse, IntelliJ, PyCharm, Visual Studio, etc)
  - ☐ Other (please specify)
- What is your preferred terminal-based editor, if any?
  - ☐ I do not use a terminal editor.
  - ☐ vim (or neovim)
  - ☐ emacs
  - ☐ nano
  - ☐ Other (please specify)

B.2 Likert scale questions

The following two questions varied from “no experience” to “very experienced” on a scale from 1 to 5, respectively.

- Rate your proficiency with the command line before CS 126.
- Rate your proficiency with the command line after the terminal assignments.

The following questions varied from “strongly disagree” to “strongly agree” on a scale from 1 to 5, respectively.

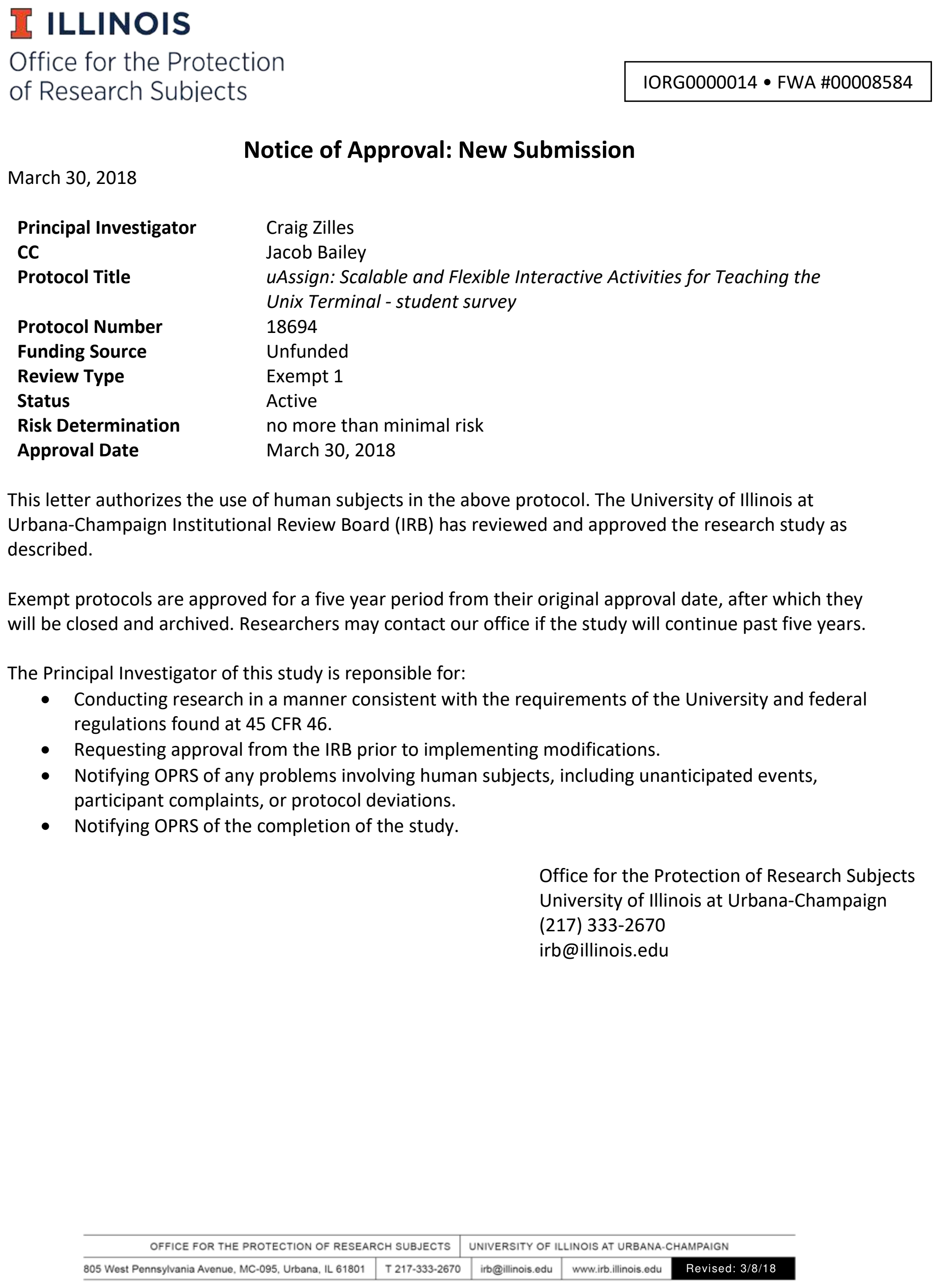
- My time was well-spent completing terminal assignments.
- I believe terminal skills are useful.
- I am confident that I can use common terminal utilities.
- I am confident that I can read terminal utility “help” documentation (`-h`, `-help`) and man pages.
- I am confident I will know the right terminology to ask questions about issues I have in the terminal.
- The terminal assignment system provided a friendly user interface.
- The terminal interface provided on PrairieLearn was bug free.
- I prefer to have terminal assignments in PrairieLearn as opposed to a different website/homework system.
- I prefer dedicated editors/IDEs such as Atom and IntelliJ over terminal-based editors like vim and emacs.
- I prefer performing common tasks (like copying/opening/deleting files) in a GUI, rather than in the terminal.

B.3 Open ended question and optional reward

The survey also asked the open-ended question: “If you have any comments about the terminal assignment system or your experience learning to use the command line, please provide them in the space provided below.”

Optionally, students were given the option to identify themselves by providing their NetIDs to enter a raffle of ten $10 Amazon gift cards as a reward.

Appendix C IRB approval for student survey



ILLINOIS
Office for the Protection
of Research Subjects

IORG0000014 • FWA #00008584

Notice of Approval: New Submission

March 30, 2018

Principal Investigator	Craig Zilles
CC	Jacob Bailey
Protocol Title	*uAssign: Scalable and Flexible Interactive Activities for Teaching the Unix Terminal - student survey*
Protocol Number	18694
Funding Source	Unfunded
Review Type	Exempt 1
Status	Active
Risk Determination	no more than minimal risk
Approval Date	March 30, 2018

This letter authorizes the use of human subjects in the above protocol. The University of Illinois at Urbana-Champaign Institutional Review Board (IRB) has reviewed and approved the research study as described.

Exempt protocols are approved for a five year period from their original approval date, after which they will be closed and archived. Researchers may contact our office if the study will continue past five years.

The Principal Investigator of this study is reponsible for:

- Conducting research in a manner consistent with the requirements of the University and federal regulations found at 45 CFR 46.
- Requesting approval from the IRB prior to implementing modifications.
- Notifying OPRS of any problems involving human subjects, including unanticipated events, participant complaints, or protocol deviations.
- Notifying OPRS of the completion of the study.

Office for the Protection of Research Subjects
University of Illinois at Urbana-Champaign
(217) 333-2670
irb@illinois.edu

OFFICE FOR THE PROTECTION OF RESEARCH SUBJECTS | UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN
805 West Pennsylvania Avenue, MC-095, Urbana, IL 61801 | T 217-333-2670 | irb@illinois.edu | www.irb.illinois.edu | Revised: 3/8/18

Appendix D uAssign documentation

This appendix contains uAssign's markdown documentation, converted for print. This includes some extra information about how uAssign can be run locally along with longer examples, but may contain information that has already been stated.

D.1 uAssign

uAssign (`ua`) is a system which allows terminal access to docker containers over websockets, based on specifications. The uAssign server manages the full lifetime of the containers, including building, running, and cleanup.

D.1.1 How it works, end to end

0. The service is setup. The client and server share a symmetric key, which will be used to keep assignment information (like secrets/answers) away from a student.
1. A client builds a specification, and encrypts it using the shared key. This specification includes information the server can use to build the image. This information can vary depending on the requirements. The encrypted data can be handed off safely to another user.
2. A client (not necessarily the same as in part 1) sends the encrypted specificiation to the server in order to create an "instance" (an instance of a specification, with an associated container). The server produces the instance by taking the encrypted specification and combining it with scripts/templates, then building a docker image and container.
   - If, when given a specification, the server already has an instance of that specificiation, it will kick any users of that instance off (more later), and reuse that instance.
   - If there isn't an active instance, a new one is created, which consists mainly of a new docker image and container. When not in use, the container is stopped on the server (and removed after some time).
3. The server gives the client back the instance's ID. The client now connects to the server over a websocket, providing that instance ID.
   - If the instance isn't running, then the container is started.

    - If the instance is running, then the other connection is closed, and the incoming connection takes over.

4. The server then runs a command on the container (specified in the spec), and proxies stdin/stdout/stderr over the websocket (using a modified terminado protocol).

5. When the client closes their connection, or no activity is seen for some time, the container is stopped. This container can be returned to later.

   Additionally, a client can ask the server to clean up an instance, removing it from the server. This can be used to preemptively clean up (for example, when the instance is known to no longer be needed), or to reset things to a fresh state to begin again.

The server manages instances over time by keeping track of their use. Once an instance (and its image/container) are no longer needed, they will be removed from the server. If a user requests a given instance again, it will fail. However, a user can re-send a specification, and obtain a new instance.

D.1.2 PrairieLearn integration

All of the work that involves a "client" is currently done through PrairieLearn, though that's not a strict requirement. See `prairielearn.md` for more info.

D.2 Running uAssign locally

Running uAssign locally requires docker (of course), and docker-compose. So far, uAssign has only been tested running on *NIX systems, and should work as expected on both Linux and OS X.

D.2.1 Local server

First, configure the environment variables by populating `.env` in the root directory of your clone of this repository:

```
UA_DEBUG=true
UA_DATABASE=postgres://postgres:mysecretpassword@localhost:5432/postgres?sslmode=disable
UA_AES_KEY=<KEY HERE>
UA_DISABLE_LIMITS=true
DC_ASSIGNMENT_PATH=/home/jake/pl/ua-cs126/assignments
```

This enables debug logging, postgres info for a local container, no limits on containers, as well as the path to the assignments. `DC_ASSIGNMENT_PATH` sets the path on the host to the assignments directory, which will be mounted as a volume.

`UA_AES_KEY` is a base64 encoded 128-bit AES key used for spec encryption between PL, the client, and uAssign. This key is known to PL and the uAssign server. One can be generated by doing:

```
$ openssl rand -base64 16
```

Once `.env` is populated, run `docker-compose`:

```
$ docker-compose up
```

This starts both the database and the uAssign server, and will quit on an interrupt. The HTTP server is available at port 8000.

D.2.2 Local PrairieLearn

PrairieLearn is used as usual, however, the uAssign aspects need some modification to point to a local server. In your course's PrairieLearn content repository's `elements/uassign_terminal/uassign_terminal.py`'s render function, force:

```
host = 'localhost:8000'
insecure = True
```

And in `serverFilesCourse/uassign.py`'s clean_spec function, replace the URL, like:

```
req = Request('http://localhost:8000/spec/clean')
```

Now, PrairieLearn will contact the local server. Be sure to not commit these changes with the rest of your updates.

As shown in `prairielearn.md`, it's possible to set these variables in the element itself, however it's easier to set these globally than to change every question.

D.3 Assignments

D.3.1 Directory structure

Assignments are held in a single directory. They are referred to by name. Assignments can be nested, like:

```
assignments
├── 00_basic_use
│   └── Dockerfile.tmpl
├── archive
│   ├── tar_compress
│   │   ├── context
│   │   │   ├── grade
│   │   │   │   └── grade.go
│   │   │   └── setup
│   │   │       └── setup.go
│   │   │
│   │   ├── Dockerfile.tmpl
│   │   └── index.js
│   └── tar_extract
│       ├── context
│       │   └── setup
│       │       └── setup.go
│       ├── Dockerfile.tmpl
│       └── index.js
├── compile
│   ├── c
│   │   ├── context
│   │   │   └── main.c
│   │   ├── Dockerfile.tmpl
│   │   └── index.js
│   └── c_o
│       ├── context
│       │   ├── grade
│       │   │   └── grade.go
│       │   └── main.c
│       ├── Dockerfile.tmpl
│       └── index.js
...
```

For example, to refer to the `tar_extract` assignment, you can refer to it as `archive.tar_extract` when creating a spec.

An assignment directory consists of:

- `index.js`, a CommonJS-style module. This module must export a `generate` function, which accepts a JavaScript object with the spec data, (that comes from the `data` object from the PrairieLearn question), and returns another JavaScript object with the generated instance information.

- `context`, an optional directory which is the build context given to the docker daemon. This directory behaves like a regular docker context, respecting `.dockerignore`, with the difference that it doesn't contain a Dockerfile. If this directory doesn't exist, then an empty context will be used during build.

D.3.2 Creating an assignment

Create a directory for the assignment. Then, create an `index.js` file in that assignment directory. An example of an `index.js` file:

```
var _ = require('lodash');

exports.generate = function(data) {
    var secret = data.secret;
    var filename = data.filename + ".c";
    var cSource = readFile("./context/main.c");
    return {
        imageName: "jakebailey/ua-cs126-docker:clang",
        init: true,
        postBuild: [
            {
                action: "append",
                user: "root",
                filename: "/usr/include/stdio.h",
                contents: "char _[] = { "
            },
            {
                action: "exec",
                user: "root",
                cmd: ["sh", "-c", "printf '" + secret
                    + "' | od -A n -t d1 | "
                    + "xargs -n1 printf '%d, ' >> /usr/include/stdio.h"],
            },
            {
                action: "append",
                user: "root",
                filename: "/usr/include/stdio.h",
                contents: "0 };\n"
            },
            {
                action: "write",
                user: "student",
                filename: "/home/student/" + filename,
                contents: cSource
            }
        ],
        user: "student",
        cmd: ["bash"],
        env: ["FOO=BAR"],
        workingDir: "/home/student"
    }
};
```

This example shows a number of capabilities of the assignments. Listed, they are:

- Modules can be included with `require()`, as in NodeJS, allowing the use of common JS libraries, or including modules in other files (for now, only within the assignment directory).
- The `readFile` function reads files within the assignment directory, as strings.
- `btoa` and `atob` allow for base64 encoding and decoding, respectively, as in browsers.

Most importantly, the `generate` function returns a JS object which describes the instance, including the base docker image needed, what to do after the image is built, as well as the command to expose to the user over the eventual websocket.

- `imageName` is the name of an image to begin with. In this example, this is an Alpine Linux image with some sane defaults as well as the clang compiler. If this image isn't present on the server, it will be pulled the first time it is needed. Alternatively, the `generate` function can set `dockerfile` instead of `imageName`, which is a string which contains a Dockerfile that can be used to build a custom image (rather than using something prebuilt).
- `init` enables running an init system in the docker container, which can manage zombies. The default is currently `false`, but may be changed in the future.
- `postBuild` is a list of "actions", which will be run on the container after it is built (before the user ever gets access). This offers a more performant alternative to using the docker build system.
- `user`, `cmd`, `env`, and `workingDir` set the environment that the user will have access to. Generally, this is some non-root account, in a shell, in some directory (likely home).

The `postBuild` attribute is a list of "actions", which perform various tasks on the container. All actions accept both `user` and `workingDir`, to manage which user the container will think is doing the action. Currently, three actions are supported:

- `exec` runs a command on the container. `cmd` sets the command, argv style. `env` sets environment variables. `stdin` is an optional string which will be fed into the command over stdin. The server will run the command, and will error out if the command fails (non-zero exit code).
- `write` writes a file. `contents` is a string with the contents of the file to be written. You can also set `contentsBase64` to `true`, indicating that `contents` is a base64 encoded string it should decode before writing. `filename` is the path to the file to be written. This action doesn't manage permissions, nor ensures that the directory structure allows the file to be written. You should use an `exec` action to create directories, and to change permissions if needed. Largely, permissions can be managed by changing `user` to be the intended user.
- `append` is exactly the same as `write`, but instead appends to the specified file.
- `gobuild` builds Go binaries. `packages` is a list of Go packages to build, for example, `["grade"]` for the package (and binary) called `grade`. `ldflags` is a string that is added to the Go compiler's ldflags during build (mainly to set variables via the linker, without editing the source code on the fly). Currently, the assignment directory must contain a directory called `gosrc` which is considered to be `$GOPATH/src`, but this may change in the future. Additionally, it may be more helpful to be able to templetize the Go source instead of using the linker hack to add info, but that will wait until the future.

- `parallel` executes many subactions in parallel. `subactions` is a list of actions. This is useful when needing to do many independent actions, like writing many files that don't require each other.
- `ordered` executes many subactions in sequence. `subactions` is a list of actions. Paired with `parallel`, this can be used to construct more complicated parallel configurations.

`index.js` is run each time a spec instance is created. This means that `index.js` can be changed on the server without needing to remove cached data. Running the `generate` function takes so little time compared to managing docker itself that this isn't a big problem.

All JS code that the server runs is protected against things like infinite loops with timeouts.

D.3.3 Legacy assignments

In older versions of uAssign, image building was controlled purely through templetized Dockerfiles. The data sent with the specification is used as the rendering context for the template, which is then sent to the docker daemon as normal. If `index.js` isn't present in an assignment directory, then the server will attempt to use the legacy method instead.

D.4 PrairieLearn integration

uAssign's (current) primary frontend is PrairieLearn, consisting of:

- A python library for `serverFilesCourse` which abstracts away the spec's representation.
- A v3 element (`uassign_terminal`), which renders the terminal for the browser.

D.4.1 Python library for question generation

uAssign questions are done in PrairieLearn using the v3 question type. The `uassign` library can be imported to generate specifications, encrypt them, as well as invoke cleanups on the uAssign server. For example:

```
import uassign

...

def generate(data):
    assignment_name = data['options']['assignmentName']
    filenames = data['options']['filenames']
    filename = random.choice(filenames)

    ...
```

```
spec = uassign.create_spec(assignment_name, {
    'files': [
        {
            'filename': filename,
            'contents': contents,
        },
    ],
})

data['params']['spec'] = spec
data['params']['filename'] = filename
```

In this example, we import the `uassign` Python library, then use it to create a specification, which is just a dictionary-style object containing information the server can use to instantiate a spec (for example, some number of files and file contents). The library returns a string, which is sent to the server when working with the spec. `assignment_name` is the name of the assignment known by the uAssign server, for example, `archive.tar_extract`.

The string is a JSON object, which includes an encrypted payload and associated HMAC. The server will verify the payload, and pull the encrypted object out via JSON. Currently, the AES key used to encrypted the payload is hardcoded in the `uassign` library, but this may change in the future.

D.4.2 `uassign_terminal` element

The `uassign_terminal` element handles the rendering/management of the terminal. In `question.html`, this only requires adding:

```
<uassign_terminal spec_name="spec" />
```

Where `spec_name` is the name of the spec data in the `params` dictionary as shown above. This mirrors other elements which read parameters.

In addition to `spec_name` (which is required), `uassign_terminal` has other options:

- `host`, which changes the hostname of the uAssign server, for example: `host="uassign.example.edu"`
- `insecure`, which when set to true (`insecure="true"` or similar) will use normal websockets instead of secure websockets.