**Independent Project 02**

**Requirements and Architecture**

**Ip Chan**

**City University of Seattle**

chanip@cityuniversity.edu

**Collaborative Online judge**

# **Background**

The collaborative online judge is a full-stack web app supporting collaborative online code editing, compiling, execution and result judgement.In this project I would discuss how I would design a collaborative online judge system. In order to realize this project, this research first elicits engineering requirements and then maps the generated requirements to the system architecture and high-level design.

# **Requirements**

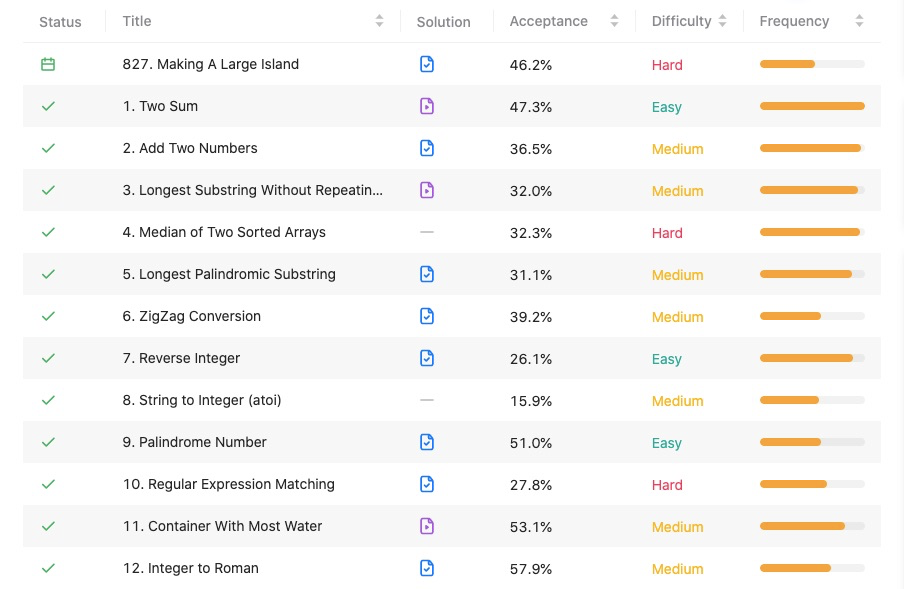
Requirements engineering is the process of defining, documenting, and maintaining requirements in the engineering design process. In this project, I have divided the requirements into two aspects (function requirements and performance requirements) and I will discuss each one in depth.

## Functional R​equirements

In this section, we will discuss in detail the functional requirements. For this course project, we will focus only on the core functionalities, and will provide detailed design and implementation plans for those core features. In the meantime, we will also briefly discuss other optional extensions, without detailed designs.

### CRUD the problems

The first core feature for this website is to allow an admin to *create, read, update, and delete* (CRUD) coding problems into the system database. Figure 1.1 shows a sample screenshot (taken from the popular online coding judge website leetcode.com) for a *user interface* (UI) for the list of available coding problems.

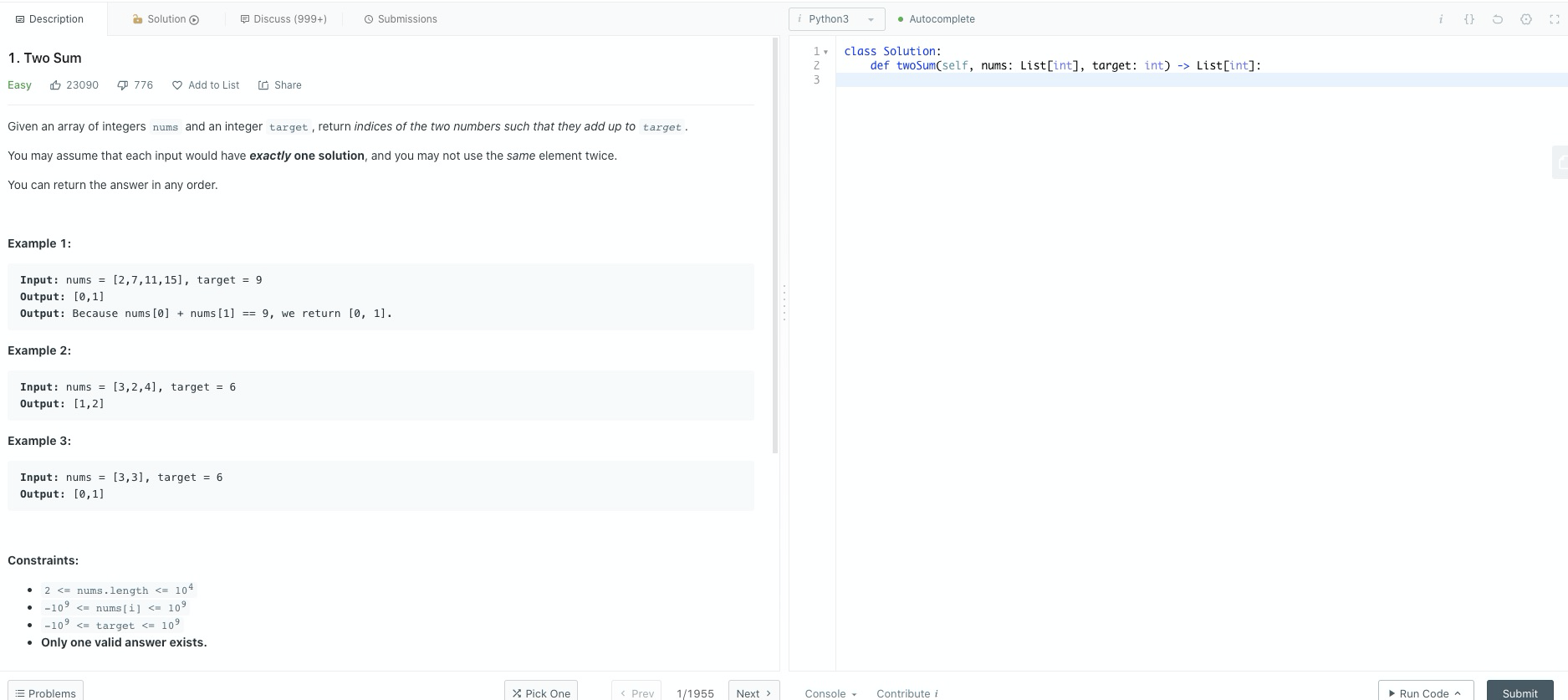


*Figure 1.1*

Our online coding judge website will be designed and implemented in a similar way. We will provide a few columns to show the problems list. The *status* will show whether the user has successfully finished each specific problem. *Title* indicates the problem itself, with its number in the database. *Solution* is an optional feature for our website, which indicates whether each problem has an official solution, and provides the link if it does. *Acceptance* rate is an optional feature, showing the total acceptance rate for each problem, which also indirectly implies the difficulty of that problem. *Difficulty*, as its name indicates, shows how difficult each problem is. Finally, the *frequency* is how often each problem is likely to be asked during a real coding interview; it’s also an optional feature.

CRUD itselfEditing co has a separate UI for admin, which is only visible by the site admins. It’s UI is very different from the one in Figure 1.1. The UI can be a relatively simple one, as it is hidden from the users.

Figure 1.2 is a sample screenshot (taken from leetcode.com) showing the editing feature for the website. On the left side, there is the detailed problem description, which was added during the CRUD feature discussed above. On the right side, users can edit the code using different programming languages. Main languages such as *C, C++, Java, Python*, etc. will be supported, with corresponding syntax highlighting. The editing area will also provide a starting template for the user.

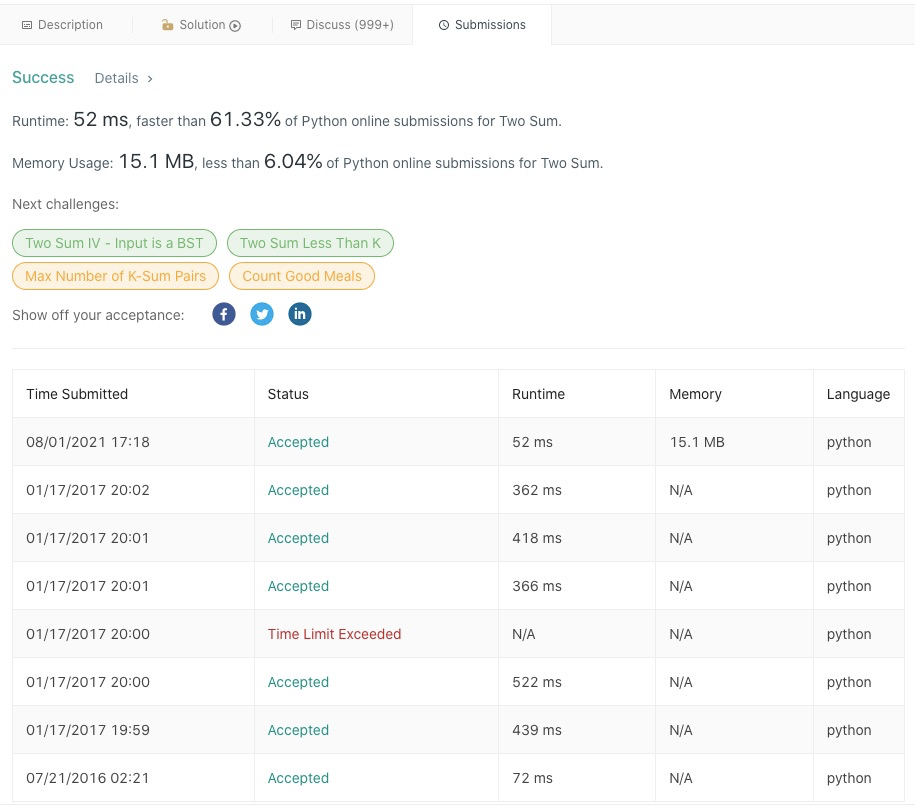


*Figure 1.2*

The main difference compared with the regular online judges such as leetcode.com is that our site supports multiple users editing the code *collaboratively*. One can imagine this is a combination with Google drive and regular online coding judge websites.

### Submit code, execute code and show results

After users have finished coding, one of them can hit the submit button. The website will then lead the users to another page showing the result of the submission, as shown in Figure 1.3. The result page will show a few important things, out of which the most important one is the result of the users’ code. If the code successively solves the problem, the page will print a big “*Success*” on the screen, and optionally provide some statistics such as the runtime and space complexity compared with other users in the world. The page also lists all the historical submissions for this problem.



*Figure 1.3*

## Non-functional Requirements

Now let’s turn our attention to the non-functional requirements of the website. These requirements are more generic than functional requirements for different projects. To simply put, there are three major non-functional requirements for most web-based applications: *security, availability, and performance*. For this course project, we will not discuss security as it generally requires expertise of its own, other than the skills that we learned from the software engineering course. However, we will provide a detailed design to scale out our website for high availability and performance, as the popularity of our website increases.

### User estimations

Since technical interviews these days are very common, out of which coding interviews would take a high percentage, we expect that our website, once getting enough attention, will be more and more popular in the near future. Let's suppose that our website will have an approximate *daily active users* (DAU) of 100,000. This number will grow in three years to about 1,000,000.

### Lookup problems

For a DAU of 100,000, we assume that each user will look up an average of 3 problems per day. As a result, we will have the following estimations for problems lookup:

* Lookups per day: 100,000 \*100%(function usage)\*3(function frequency) = 300,000
* Lookups per second : 300,000 / 86400 = 3.5
* Peak-hours lookups per second : 3.5 \* 10 = 35, assuming the website will take 10 times more hits than its normal hours.

### Editing

For a DAU of 100,000, we assume these users will be online together in the worst scenarios. We now get the following numbers:

* Average concurrent users editing the code: 100,000 \* 60 \* 60 / 86400 = 4166
* Average editing function hits per second: 4166 \* 50%( function usage) \* 10 (function frequency) = 20,000. We assume 50% of the users can type once per second. We also assume that, due to collarations, there are 10 messages received and sent by our backend servers per second; one user types a word, which is received by our backend servers, and then our servers will send this message to other collaborative users in real time.
* Peak-hour editing function hits per second: 20,000 \* 10 = 200,000, assuming the website will take 10 times more hits than its normal hours.

### Submit Code

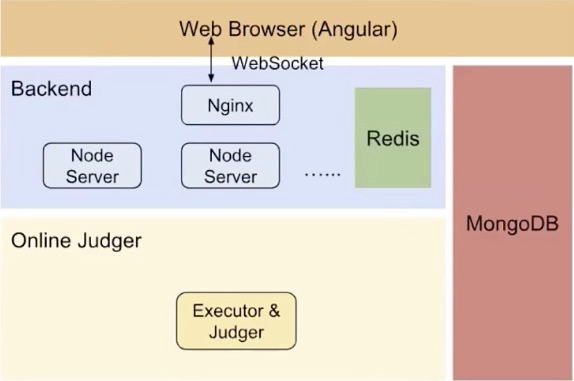
We now discuss the code submission feature, as follows:

* Function hits per second: 4166 \* 1%(function usage) \*1(function frequency) = 40
* Peak-hour function hits per second: 40 \* 10 = 400, assuming the website will take 10 times more hits than its normal hours.

Once we have some estimates, we need to think about high availability and performance of our website, once it gets more and more popular. High availability means that no matter how many concurrent users are using our website, it cannot crash in any way. Some *service level agreement* (SLA) requires 99.999% of availability, that is less than 1 hour downtime in the entire year. On the other hand, high performance means our website should perform quickly, in terms of latency, as the user number grows. It is not acceptable that our website becomes slower when the number of users increases. We will discuss how we achieve these two non-functional requirements in the next section.

# **Services and Architecture**

We can now start our high level design, including the listing of each micro-services and a high level system architecture. Figure 2.1, shows the high level architecture of the entire website, which includes the *frontend* (web browser), *backend*, communication protocol between frontend and backend i.e. *websocket*, *online judge agents*, and *database*. We will now discuss each one in detail.

****

*Figure 2.1*

## Web browser as the frontend

Frontend mainly contains the UI pages that we discussed above. These pages are implemented by technologies such as *HTML, CSS, and Javascript*. These days, engineers will likely use some third-party frameworks to write frontend code. One of these based on Javascript is called Angular. Angular is a popular framework that allows users to write highly efficient frontend code that will be compatible with many backend services. As a result, we will also use Angular as our frontend technology. Other popular frameworks include React, which also is based on Javascript. However, we will choose Angular basecuse of its more features and flexibility.

## Websocket

One of the core features that our website supports, while others don’t, is the simultaneous collaboration. To implement this feature, we need a network protocol that allows servers and clients to send to and receive from each other at the same time. One such protocol is websocket. With websocket, once a user types something in the code editor, that message will be sent by the client to our backend servers, and then our servers will send this message immediately to all other collaborative users in real time.

## Backend services

The backend services mainly will support the following features or business logics:

* Read coding problems from the database, and return them to the frontend.
* Receive messages from our users when they are typing in real time, and send the messages to other collaborators immediately.
* Interact with database to perform CRUD operations for system admins.
* Interact with database to return historical results for each coding problem to frontend.
* Interact with the online judger, and return the judge results back to frontend.

All of the above business logics will be implemented using a technology called *Node.js*, which is based on Javascript. There are many backend frameworks other than Node.js. The reason we choose Node.js is that we can use Javascript for both frontend (Angular.js), and backend. Node.js also provides good performance thanks to its event-driven mechanism. However, one can implement backend services using any well-written frameworks or technologies. We will use multiple Node.js instances to scale out our website, as mentioned in the last section.

In order to scale out our website, we will also use *Nginx* in front of the Node.js servers to perform load balancing. Nginx is a powerful tool for various purposes, including *load balancing, SSL termination, caching, reverse proxying*, and so on. Nginx can normally support *C100000* concurrencies, thanks to its event-based design. We will likely use two Nginx instances, one primary and one backup for production. We should have well-defined *metrics for monitoring and alarming* during runtime, for critical services such as Nginx, so that once the primary Nginx has any issues, backup Nginx is kicking off, *oncall* operators will be notified in time, and fix the issues for the primary Nginx. Once the primary Nginx comes back in life, it now becomes the backup, as the original backup is now serving as the primary Nginx.

Another technique to scale out our website is through *caching*. Nginx itself can do some web-caching, also called file-system caching. For example, Nginx can cache the coding problems, and returns to its client (the frontend) without talking to its upstream database. Another type of cache is called in-memory cache. This type of cache is much faster than the file-system cache. *Redis* is a popular choice. Other options include *MemcacheD*. We will use Redis in this project as it provides more features. Redis can cache, for example, the coding problems, the messages collaborative users type in real time, and so on.

## Database

As for the database, it will store the following information:

* User profile
* Coding problems
* Coding results for each problem for each user
* Optionally, the statistics for each coding problem, such as the average acceptance rate and so on.
* Potentially others

There are many debates whether to choose *relational and nonrelational* databases. It really depends on each particular situation. For our project, we will use *MongoDB* as the main database. It is a nonrelational database, at the same time it also supports CRUD and search operations efficiently. We might end up with a different database later, depending on our scalability of the site.

## Online judges

Online judges are a set of background processes that take coding submissions from the backend servers, and perform a judgement to decide whether that submission passes all the test cases. Therefore, online judge needs to perform the following logics:

* Build and compile the code submitted by the users
* Run the compiled code against all the test cases (which was also CRUDed by the system admin)
* Return the result back to the backend services.

Considering that online judges will have a huge load if our website gets more popular, we should be using some high performance programming language to implement it. C/C++, and Java are good candidates. Python is also good due to its simplicity. On the other hand, the underlying architecture for online judge clusters is more important. We will use multiple instances of online judges to scale out our website.

Nginx, Node.js servers, Redis, Online judges and MongoDB are all critical services for our website. We need to have well-defined metrics for runtime monitoring and alarming, including availability and performance metrics. For example, if the number of Node.js that is down has been bigger than the threshold, the alarm will go off, and the oncall operators will get engaged. Another example of metrics is that if the average latency or P99 latency of the Node.js services is greater than the threshold, the alarm should also go off.