# Design Log

## For starters

The first hurdle to get past was how the software should behave. This depends on its outward complexity. If it is nothing more than a simple one target load tester all it would need to do is launch once and run the test.

However, it would be nice to run multiple tests, and further to that it should know about all the tests that are currently running (along with any other test specific data it may have). This means that it’ll need a single running background application that can track what’s running. The best way of achieving this is through a daemon/service model.

This complexity then gives us more options for what it can now do; certainly regarding its outward API. With the program running as a service it gives it a certain level of innate security not offered as a single program as it allows the software to be relaunched on shutdown, potentially even restarting tests that didn’t finish.

It can also allow for different applications/front ends to talk to it and either control the software or simply query it for information. The control aspect is of particular importance as it will allow proper clean up if a test is cancelled instead of potentially being forced to ‘kill -9’ it.

## What runs where and how (thread/tasks)

After deciding that it should run as a daemon, the next step was to Figure out the best way of handling commands and state changes. Along with making sure that data can be passed between all relevant points of the program to where it is needed.

The original ACP diagram to model each concurrently running thread had to be updated to A) use the ACP diagram correctly, B) reflect newly gained knowledge regarding the API and, C) a change of opinion in how the program should be interfaced with.

## Thread Pool Queue

When this project was first started the initial plan was to have threads spawn wherever they were required (with no attention being paid to tracking them, other than in the instances that they existed). After some research and advice I learned of the “Thread Pool Queue Pattern” (TPQ). This allowed me to have a central thread handler that simply dished out work to threads as they were required. The threads being put to sleep upon a job being completed.

Another very important component of the TPQ was that it needed to be more or less totally agnostic to the work it was doing to cut down on maintenance; so that a seperate thread spawner wasn’t needed for every possible method/task that it would be running.

## Logger

The most important thing the logger needs to do is to A) log things and to B) not slow down the program. To this end it was decided that the logger should run as a seperate task. With each component only writing to a log message queue that is then gradually flushed to the file itself.

This method should help in making sure that the program doesn’t end up bogged down in writing to the disk (a much slower task than simply using CPU). Though, processes dealing with disk writes are priortised over processes vying for CPU usage [VERIFY].

## Test Case Analyser

The test case analyser is one of the most important parts of the system. To that end making sure that its design is clear, robust and totally thorough is vital. Testcases will be written in a “human-readable format”; in this case XML. That means a parser will be needed so that the testcase can be turned into something the program itself can understand.

Once the document is parsed, its data needs to be stored in some way that can easily be accessed by the plugin and the other parts of the program that might require it.

The first iteration of the design had some serious flaws that left large parts of somewhat ambiguous and too generic/error prone, made evident before the state changes had even been placed. However, the second design left a much more robust and, hopefully, easier to extend model that can be worked with.

## API

The most appropriate way to communicate with the software would be using a proper API. The first option that came up was to use a SOAP/REST solution. However, after doing more research the gRPC project was mentioned. This allows me to use Googles Protocol Buffer system. Most importantly it allows for scalibility and layered designs for things like monitoring, logging and load balancing and is designed around duplex communication, perfect for implementing the CLI features.

Internally the service will be using a simple socket to collect requests, and inter-thread communication will be handled with queues, conditional variables and mutexes. The reason a proper API is so vital is to provide a common interface for outside applications to request data on the program in a tidy and uniform manner. It also means that only a single interface is required for all forms of communication from all possible end-users, cutting down on maintenance and possible points-of-failure.

Making sure data is collected and passed up through the program to be correctly displayed will be a challenge during development, however, along with designing the classes with APIs in mind.

To this end I will likely create a library that will handle information being passed to and from the API. This means I can design my classes with nothing but functionality in mind, and instead abstract them out to a helper class that handles moving the data into a serialised format.

## Database

The database needs some sort of design, even if it won’t be implemented. To this end an Entity Relationship Diagram (ERD) was created to model what the database would/should like if it were to be created.

Due to the software having a requirement to be “cloud” based, each instance would need to know where it is in the world. These “instance locations” can host many instances of the testing software. Each instance will contain a set of previously run tests. Each test will track its own results too.

Making sure the design was codified and standard was the hardest part, having no previous tangible experience in ERDs. And for every step of progress I made, redesigns and re-planning was done to conform to what ever new Idea I had.