Odin - Functional Specification

Student 1: James McDermott **Student Number:** 15398841

Student 2: Martynas Urbanavicius **Student Number:** 16485084

Project Supervisor: Dr. Stephen Blott

Completion Date: 21/11/19

0. Table of Contents

1. Introduction

1.1 Overview

Odin is a programmable and extendible job orchestration system which allows for the scheduling, management and unattended background execution of individual user created tasks on Linux based systems. The primary objective of such a system is to provide users/teams a shared platform for jobs that allows individual members to package their code for periodic execution, providing a set of metrics and variable insights which will in turn lend transparency and understanding into the execution of all system ran jobs. Odin aims to do this by changing the way in which we approach scheduling and managing jobs.

Job schedulers by definition are supposed to eliminate toil, a kind of work tied to running a service which is manual, repetitive and most importantly - automatable. Classically, job schedulers are ad-hoc systems that treat it’s jobs as code to be executed, specifying the parameters of what is to be executed, and when it is to be executed. This presents a problem for those implementing the best practices of DevOps. DevOps is something to be practiced in the tools a team uses, and when traditional job schedulers fail they introduce a new level of toil in debugging what went wrong.

Odin treats it’s jobs as code to be managed before and after execution. While caring about what is to be executed and when it will be executed, Odin is equally concerned with the expected behaviour of your job, which is to be described entirely by the user’s code. This observability can be achieved through a web facing user interface which displays job logs and metrics. All of this will be gathered through the use of Odin libraries and will help infer the internal state of jobs. For teams, this means Odin can directly help diagnose where the problems are and get to the root cause of any interruptions.

1.2 Business Context

No business organization(s) will be sponsoring the development of this system nor the way in which the system will be deployed. Despite this fact, however, Odin itself can be deployed anywhere that offers a Linux based subsystem, such as AWS, DigitalOcean, Azure and Google Cloud. Odin can also be self-hosted.

1.3 Glossary

DevOps ideology - a culture of collaboration and sharing aimed at bringing the software development and operations teams together to help eliminate constraints and decrease time-to-market.

Cron - a software utility cron is a time-based job scheduler in Unix-like computer operating systems. Users that set up and maintain software environments use cron to schedule jobs to run periodically at fixed times, dates, or intervals.

Time series database - a software system that is optimized for storing and serving time series through associated pairs of time and value.

Observability - a measure of how well internal states of a system can be inferred from knowledge of its external outputs.

Distributed system - a system whose components are located on different networked computers, which communicate and coordinate their actions by passing messages to one another.

Programmable Libraries - a collection of non-volatile resources used by computer programs, often for software development.

Goroutines - functions or methods that can run concurrently with other functions or methods. These can be thought of as light weight threads.

2. General Description

2.1 Product/System Functions

Odin will provide a management system based around the periodical execution of jobs written by the user. As previously stated, Odin is concerned with the conditions before and after the time of execution, so Odin also provides the observability of jobs through centralised logs and metric visualisation.

These jobs will be written in languages like Python, Golang and Node.js initially, so users will be provided language support in the form of libraries and packages in their language of choice. These libraries provide functionality that will help infer the internal state of jobs and any given time. This is how Odin is able to determine expected behaviour - each job will import it's language specific library. In this way, Odin is said to be a programmable system which is significantly extensibile in comparison to other job schedulers.

Once a job is running, we want to know it's behaviour and history of execution. This observability can be achieved through a web facing user interface which displays job logs and metrics. All of this will be gathered through the use of Odin libraries and will help infer the internal state of jobs. For users, this means Odin can directly help diagnose where the problems are and get to the root cause of any interruptions. The job logs and metrics will be disaplyed through a Web UI comprised of Node.js, AngularJS and ExpressJS.

Odin is to be a distributed system. This is a stretch goal we have set which we hope will increase the reliability of the system against single-machine failures. This could be achieved using a distributed reliable key-value store like etcd or Consul. Both etcd and Consul are written in Golang and use the Raft consensus algorithm to manage highly-available replicated logs.

In all Odin is to be a job orchestration system that supports teams and users who implement the DevOps ideology.

2.2 User Characteristics and Objectives

2.2.1 - Desirable Characteristics

The community of users we expect to use Odin are, as we mentioned before, teams and users who implement (or are actively looking to implement) the DevOps ideology. These kinds of teams have a great deal of expertise in software systems and the application domain, as said above, DevOps is something to be practiced in the tools a team uses and tools like Kubernetes, Docker and Jenkins are all commonly accepted industry standard tools used in many teams practicing DevOps. From this we are able to assume that most teams have a good deal of experience with orchestration based systems.

2.2.2 - Requirements of the User Community

When it comes to what teams should expect from Odin, the key requirement we need to satisfy is to successfully ensure the scheduling and the periodic execution of individual jobs described solely by the user's code. With this said, the wishlist does extend beyond this.

A user can expect that any failure in a job's execution is not due to any internal components of the system (such as the scheduler or the execution centre) but is found to be an error in the user's code which halts the systems from running the job. Leading on from this, these user’s will require a way to generate/view the history of execution associated with each job created, whether it is running or not. This is an important note as classical job scheduler systems make this a rather unfriendly process, namely by storing all such crucial data in log files nested somewhere on the Linux filesystem.

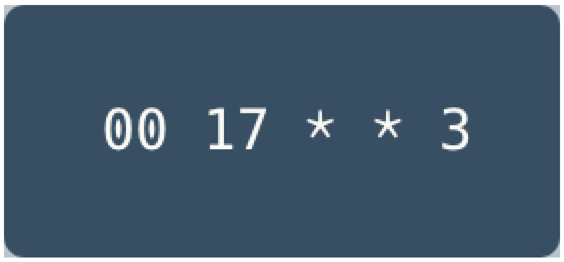
User’s would expect a less verbose scheduling format. Currently the standard version of Cron provides a five field identifier format which signifies the Minute, Hour, Day of the Month, Month, and Day of the Week on which a job should execute. This format is fine, but we believe the Odin time string format should be simpler.

Teams will need a way to makes changes to active running jobs. First and foremost Odin is an orchestration system, so further modifications and even the removal of running or stopped jobs is another vital requirement. Satisfying this requirement means individual users will be in complete control of their respective jobs after the point in time during which they were created.

2.2.3 - Satisfying Requirements

- Both the scheduler and the execution centre should perform to user expectations, and should be generally fast. This leads us to one of the reasons why we chose Golang as the language to power Odin. Golang has a built in capability called Goroutines, which are functions or methods that run concurrently with other functions or methods. Without digging too deep into the anatomy of Goroutines, they generally act like light weight threads, and the cost of creating one is miniscule when compared to a standard thread. It is common practice for Go applications to have thousands of Goroutines running concurrently. This generally means that execution of individual tasks can happen concurrently, bypassing potential issues such as greedy processes (the hogging of resources) and the creation of a task backlog.

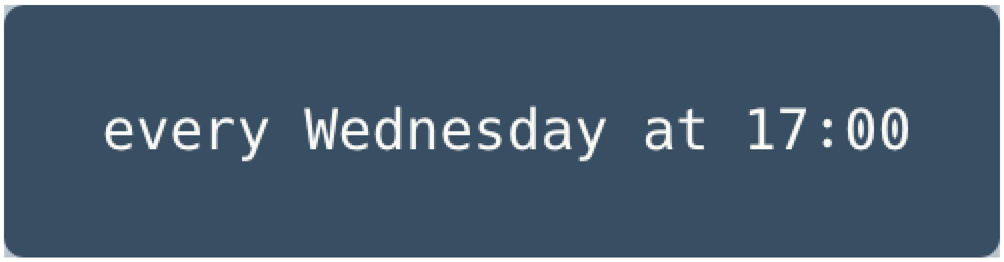
- As said, we want to move away from scheduling formats such as the cron time string format, and giving users a more simplistic format, and this is something user's should expect from such a system. Here is an example of a cron time string format for a job executing every Wednesday at 5pm:



```
00 17 * * 3
```

The cron format uses an asterisk to signify that any value will work in this place, so in the example above it doesn't matter what the Day of the Month or Month values are, as long as it's the third day of the week and it the Hour and Minute fields are 17 and 00 respectively

Let's take a look at the Odin time string format:



```
every Wednesday at 17:00
```

It's much cleaner, in fact it's significantly more readable and clear-cut. We remove the need for the asterisk as it's implied in the definition above. Let's say however, we wanted to execute on every Wednesday and Friday at 18:30, here is an example of a cron time string format that job:

```
30 18 * * 3,5
```

Converting this into the Odin time format we receive:

```
every Wednesday at 18:30 and every Friday at 18:30
```

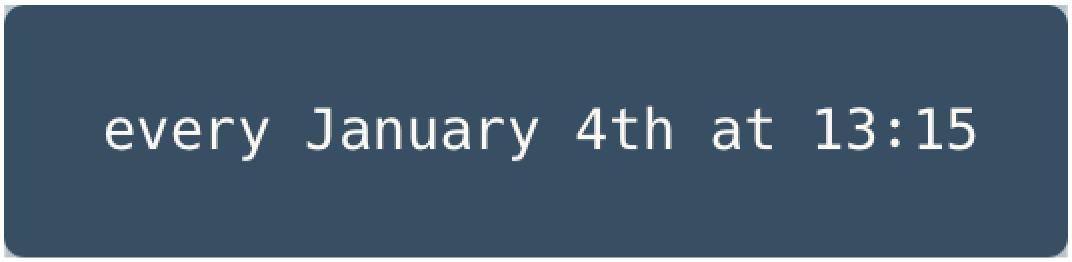
While it's a little bit longer that the cron format, verbosity is a benefit, it's better to have code or parameters Let's look at a final example of cron and Odin time formats:

This job executes every January 4th at 13:15 regardless of the day

```
15 13 1 4 *
```

of the week. Here we see the issues cron time formats run into the order make little sense. In the below example we've converted this to the Odin format:

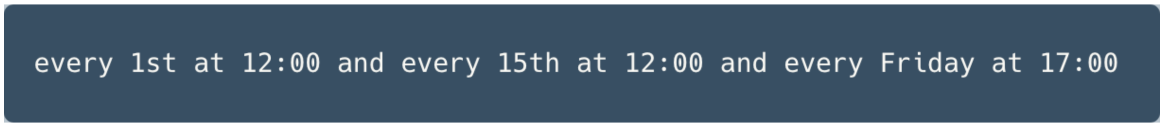
As an aside, all of the Odin formats you've seen are able to be



```
every January 4th at 13:15
```

chained together, that is to say that the below statement is perfectly usable:

In the above format, statements are chained together using the



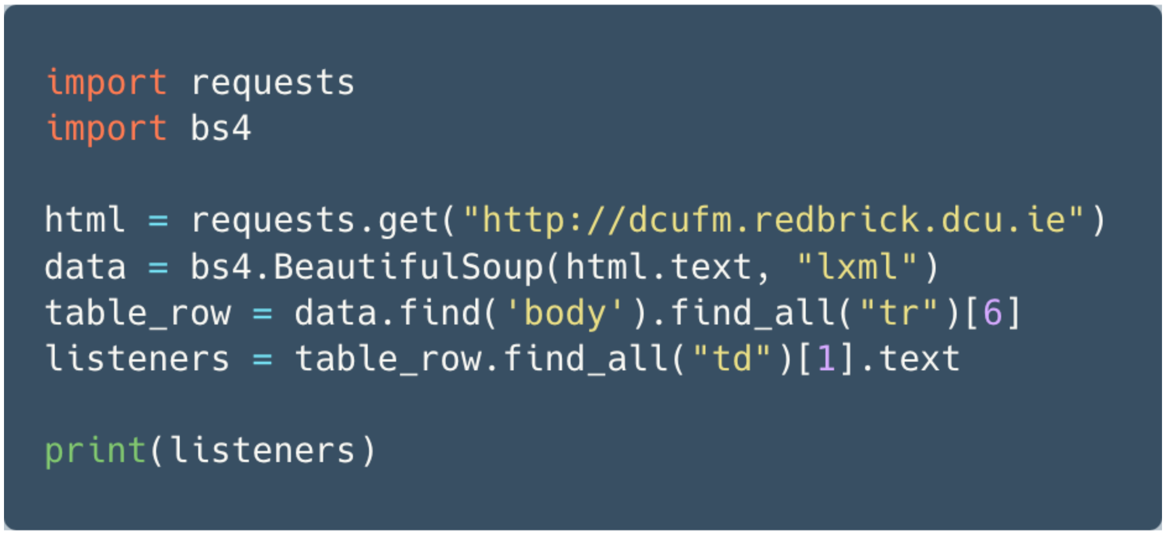
```
every 1st at 12:00 and every 15th at 12:00 and every Friday at 17:00
```

and keyword. The job executes every first of the month at 12:00, every fifteenth of the month at 12:00 and then every Friday at 17:00. This linking of times together can be especially useful in the case of running general purpose clean up or backup scripts jobs.

To ensure expectations are met we will have a rigorous testing workflow for the scheduler and execution centre during development. Testing of the code can be provided through [the standard Golang testing package](), which also has built in capabilities that allows us to perform benchmarking of each component.

- We can also provide users a view into the history of execution through a metrics collection system defined in the user code through the inclusion of a language specific library. Each library will give user’s insight into the behaviour of the code, meaning they will be given a way to view the internal state and metadata associated with each and every job.

  Take for example we have a Python3 script we wish to turn into a job:



```python
import requests
import bs4

html = requests.get("http://dcufm.redbrick.dcu.ie")
data = bs4.BeautifulSoup(html.text, "lxml")
table_row = data.find('body').find_all("tr")[6]
listeners = table_row.find_all("td")[1].text

print(listeners)
```

  Importing the Odin Python3 library we can transform the code into a job like so:

```
import requests
import bs4
import odin

odin.config("scrape_dcufm.yml")

html = requests.get("http://dcufm.redbrick.dcu.ie")
if odin.condition(html.status_code != 200):
    odin.result("HTTP request returning a non-200 code")
else:
    odin.watch(html)

data = bs4.BeautifulSoup(html, "lxml")
odin.watch(data)

table_row = data.find('body').find_all("tr")[6]
odin.watch(table_row)

listeners = table_row.find_all("td")[1].text
odin.watch(listeners)

odin.result("There are " + listeners + " people listening.")
```

Let's break down the above file piece by piece. We import the library we've written "odin" with the *import odin* statement. This gives us access to a set of methods. We can set the config Odin uses to *dcufm_scrape.yml* by default Odin will search for *odin_job.yml*. This configuration file is a YAML file which used to describe and calculate a jobs metadata (name, time running, time format, etc).

From here we see two other Odin functions in *watch()* and *result()*. The *watch()* method simply tells Odin to take note of the value specified - this special logging capability is a fundamental part of Odin, this is where the observability happens, with values stored for use in the user interface where metrics and all logs are centralised. Finally, the *result()* function is used to infer the end state of the job. It is called twice in the program above, but only ever executes once. In this way it acts like a return statement in a function, ending execution upon being reached.

These are of course only a subset of some of the functionality found in the odin library. We aim to provide methods that allow for the testing of a job before it's addition to the system. The method *mock()* will allow such functionality. We also hope to give users to configure their own storage options for values (such as InfluxDB, Prometheus or other time-series databases).

- Providing users with Odin as an orchestration system means designing Odin to reduce the time and manual manipulation required to align jobs around a team's applications, data and infrastructure. Odin simply arranges tasks to optimize user defined jobs as a workflow, meaning jobs can be modified and further altered after their point of execution. A user will be provided a command line tool in odin-cli or simply - odin, to interact with this workflow. Here is a sample of Odin's user guide, the output of odin --help. This shows the commands available to Odin and their purpose:

```
orchestrate your jobs for periodic execution

Usage:
  odin [command]

Available Commands:
  deploy      deploy Odin jobs
  describe    describe a running Odin job
  generate    creates config files for an Odin job
  help        help about any command
  list        list the user's current Odin jobs
  log         show metrics and logs associated with Odin jobs
  modify      change details about an Odin job in-place
  remove      remove an Odin job
  status      return the status of an Odin job
  upgrade     upgrade Odin to the latest release
  version     show current version of Odin

Flags:
  -h, --help   help for odin

Use "odin [command] --help" for more information about a command.
```

2.3 Operational Scenarios

2.3.1 - Pruning Docker containers

When working with Docker, containers, networks and images often all go unused as containers are spun up on a whim. Over time this can build up and consume gigabytes of storage. This job aims to automate the pruning of such things. It's a process the user shouldn't have to deal with. This Odin job will remove all dangling images and volumes. We want this Odin job to execute at the end of every workday so resources created during each day are removed.

A user defines this job with the following code:

```
import os
import odin
odin.config("docker_prune.yml")
x = os.system("docker system prune --volumes")
odin.watch(x)
if odin.condition(x == 0):
    odin.result("Prune successful!")
else:
    odin.result("Prune failed - error: " + x)
```

Along with this autogenerate YAML config *docker_prune.yml*:

```
provider:
  name: "odin"
  version: "1.0.0"
job:
  name: "docker_prune"
  description: "This jobs prunes dangling Docker images, networks and volumes."
  language: "python3"
  file: "docker_prune.py"
  runs: "everyday at 17:30"
```

In the directory where these files exist, the user simply executes the following on the command line:

odin deploy -f docker_prune.yml

The YAML file is parsed to build the object that defines the job, this contains the job name, the job description, the amount of time it's been running, the unique hash associated with the job and the scheduled execution times. In the case above, Odin engine 1.0.0 is specified to be used and the job's language is defined as Python3.

The schedule table is built using the runs parameter and the file specified in the job section of the YAML file gives the path to the code used to describe the job. The contents found at the end of this path are then copied to be under /etc/odin/jobs/. They can be found in a directory named by a unique hash generated by the Odin engine. This directory contains the YAML and Python files, and uses this absolute path for execution of these files.

Once this has all taken place, values and logs from the job are extracted upon each and every execution and this is inserted into a time series database. These can be used build Odin's user interface, giving the executing user an insight into the inner state of this job.

2.3.2 - Keeping databases up to date

During his INTRA placement James worked on a large redesign project for which the databases were shared with a data migration team. This team would apply SQL scripts to update any and all aspects of the databases, renaming tables, modifying the schema, dropping rows or columns, etc. James worked on the building, and deploying of the code for the QA teams and often when carrying out this work would stumble into an issue with the database on his local machine being out of date due to a high frequency of scripts being run in since his last refresh.

Refreshing to the latest version could often take anywhere between 10 to 15 minutes. This could easily have been executed as a background process. In this case, a fix to the issue would have been the existence of a user defined job scheduling system which could remove this TOIL. A job would run every 20 or 30 minutes to see how far behind the local was from the remote. If it was found to be behind it would be updated. Otherwise it would do nothing and wait until it's next execution. Such a system like Odin did not exist in James' workplace.

2.3.3 - Generating tokens

During his INTRA placement, Martynas worked with a multi-cloud piece of software which required periodic updates of the access token. This token was used to specifically access certain databases which were required by the testing framework. This means that the framework was essentially useless as tests could not be run without a token which was in date.

The update of this token was required at least every month because the tokens had an expiry date of 30 days. To refresh the access token a series of requests had to be made, which could consume 2-3 minutes if caught straight away. The process could have easily been automated for each user as a wide ranging job to ensure each user was using a token which was in date, therefore alleviating any blockers to running the tests.

2.3.4 - Scraping the entire CASE4 timetable

The DCU timetable service isn't that reliable, in fact it sometimes experiences outages for a few days at a time. Many students have scrapers and tooling built around it in order to query it, but this proves futile when the service is down.

The purpose of this job would be to constantly pull the CASE4 timetable daily so that in the event the service goes down we will have the latest copy. This will be served from the host machine's /var/www/html/timetable/ directory.

This job would be executed everyday because the timetable can be updated at any time to modify the placement of lectures and labs throughout the week, and we must ensure we have the most recent version.

2.4 Constraints

2.4.1 - System Constraints

As mentioned previously - the Odin system is being developed exclusively on Linux based systems. One large disadvantage of Linux is the abundance of ways in which software can be packaged. Likely installation of Odin will require dependencies installed via four different package managers (go get, apt-get, pip, and npm) at the very least. There are ways of automating installation to reduce TOIL for the user, but Linux's lack of an absolute installation mechanism will make the installation of Odin a somewhat harrowing process. We will aim to reduce the TOIL that comes with the process.

2.4.2 - Language Constraints

Though each of the team members has used Golang for about 9-10 months, neither of the teams members have ever used Golang on a project of this scale previous to this. As this is the teams first large project with Golang it's imperative the team further improves their knowledge of the language running at scale. The team is building libraries for Odin in Golang and other languages, namely Python and Node.js (with which they each have 3-3.5 years of experience using). Despite this experience, this does put extra overhead on the team to write code in at least three different languages to support the system.

2.4.3 - Time Constraints

The project is due to be completed in mid to late May 2020. The team believe this is adequate time in which to complete the feature set they have already outlined. As mentioned earlier in this document, the team has set a stretch goal of making Odin a distributed system. This would be beneficial to Odin in regards to reliability of its core components, with jobs using resources across many machines as opposed to one. Time will be a limiting factor in whether or not this can be achieved, as the team acknowledges that distribution will introduce a whole new set of problems.

3. Functional Requirements

3.1 Perform as an orchestration system

Description	Odin as an orchestration system should perform to reduce the time and manual manipulation required to align jobs, data and the infrastructure beneath them. This requires ensuring the controlled management of resources and unattended background execution of individual user created tasks on Linux based systems whilst allowing user to deploy, remove, query, and modify jobs at any time.
Criticality	This is a fundamental part of Odin as a whole. Odin is being developed to run jobs, which in turn allows the reduction of TOIL for teams. Acting as an orchestration system enforces the idea that TOIL is something to be eliminated, Odin should require as little manual input as possible to manage resources

	and jobs for teams. It’s critical that Odin alleviates the pressure from a team by efficiently aligning a job, it’s related metadata and the infrastructure binding it together without the team having to worry about such things.
Technical Issues	The main technical challenge we may face is linking the system together. From the CLI tool to the time-series storage solution to the scheduler and execution units to the centralised logging and metrics centre on the user interface, there truly is quite a bit going on with this system. This can really be boiled down to an issue we face in managing the data in the system. We will need to build an underlying infrastructure that allows precise communication of job data, which can be achieved through shared directories and the time series database. Along with this, testing and validating an orchestration system will prove difficult. Testing the infrastructure on which Odin is built means testing the flow of data around the system. Analysing data flow is the only way we can ensure the bindings to and from each component actually exist and work. This is a technical issue we hope to address in our final solution.
Dependencies	To the team behind the development of Odin, almost all aspects of the project and further functional requirements depend on this aspect of performance. Performing as an orchestration system therefore can be said to a dependency for each other functional requirement, is it is integral to the system. Therefore the one dependency the orchestration system has is the aforementioned infrastructure the team chooses to build it upon

3.2 Perform as a scheduler and executor

Description	Odin should perform as a scheduler and executor as these are the core components of any task management or job planner system. This is defined as a requirement because if jobs are not created at the correct user-specified times, or if they are not defined for the correct recurring periods, or if the jobs do not actually execute at the expected user-scheduled time, then the system can be said to be unfit for purpose.
Criticality	Without doubt, this functional requirement to perform as a centre of scheduling and execution is highly critical to the success of this project. If jobs are irregularly executed or in some cases not executed at all, this introduces more TOIL than it actually removes. For DevOps based teams this is certainly a move in the wrong direction. If Odin is to be used for team based work then we must ensure this requirement is met due to it's crucial nature.
Technical Issues	The end-to-end testing will be important but we have acknowledged that for the scheduling and execution of a job that executes only once a month, end-to-end testing could prove tough. Small time spans such as everyday and every week can be achieved easier, but testing over large time spans is something we have to figure out. By default, we do believe a core set of unit and integration tests built around all time spans is imperative to the systems validation as a scheduler and executor of user defined tasks. This is a technical issue we hope to address in our final solution.
Dependencies	Execution and the scheduler are totally dependant on the system the team create. Likely an intermediary

	language will be made to support the simpler time string formats, and execution will happen at a lower level. The team expects that libraries such as the Golang "work" library will also be fundamental in ensuring the success of the aspect of performance.

3.3 Perform as an observability platform

Description	Odin should perform as an observability platform. While users can arbitrarily create, remove and modify jobs, we must understand that no system is 100% efficient. Due to this fact, jobs are going to fail. Odin aims to give a direct insight into the live values used in jobs so that in the instance they fail a user can easily debug the issue in their code. This can be achieved through strong structured logging.
Criticality	While this aspect of performance is not as vital to the system functioning as a job scheduler, it is something we as a team feel is important. When jobs go wrong, the user should know why, it's that simple. Building this into our code gives users a way to take the first step in fixing an error by identifying what went wrong.
Technical Issues	The greatest challenge faced in this is the creation of the observability pipeline. There must exist a way to transport data from the user's code and store it so it may be displayed on the user interface. We hope to achieve this through the use of programmable libraries which take values from the live code and then store these. Storage could be extended beyond a time series database but for now that's what we feel is the best option. From here it is just a matter of extracting the values where appropriate from the

	code powering the user interface.
Dependencies	The existence of our observability pipeline is totally dependent on its efficiency in moving values from one location to another. This performance task depends on the time series storage, and programmable libraries as key components to ensuring its success.

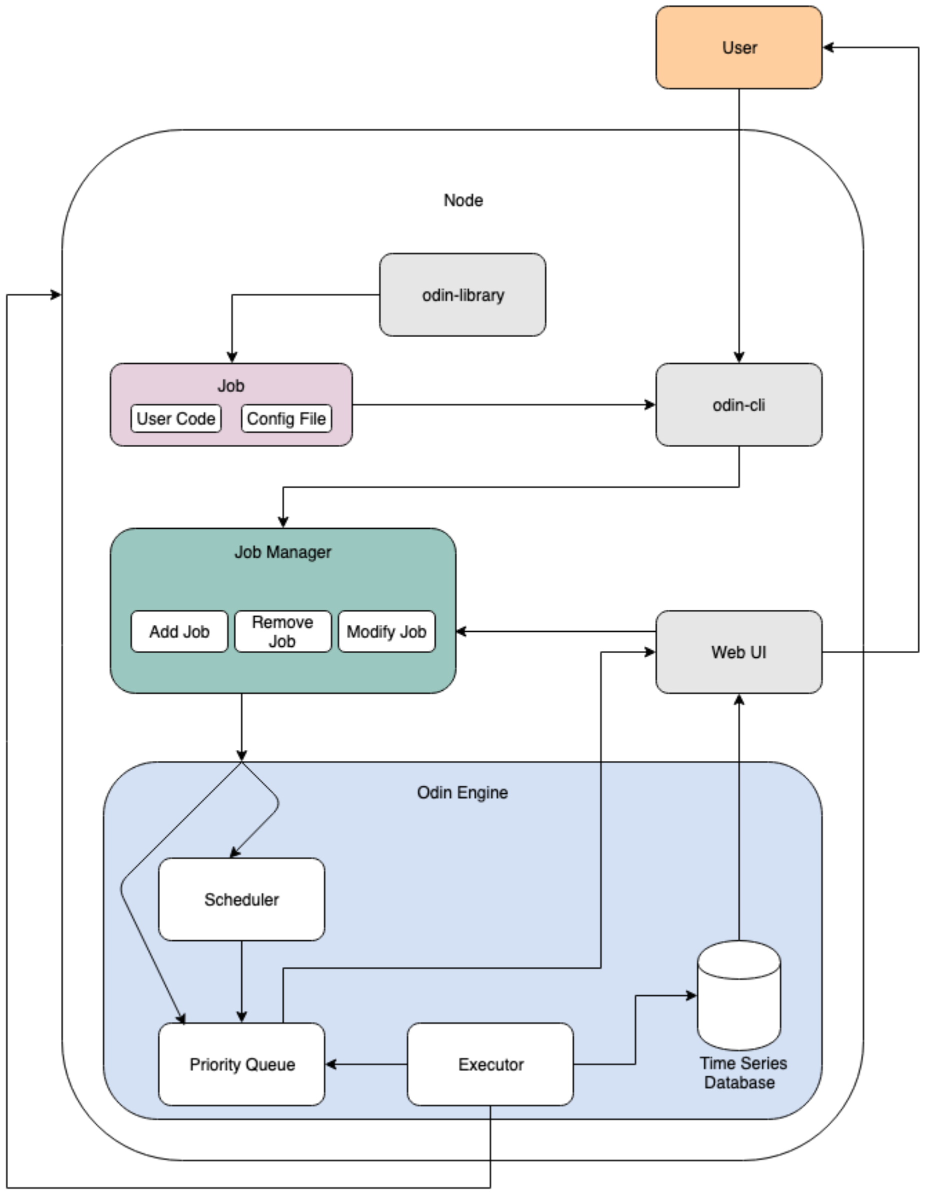
3.4 Perform as a set of programmable libraries

Description	Odin should perform as a set of programmable libraries which decorates the user's code. These will be language specific, meaning a job cannot be created for Odin if there does not exist a library to build jobs in for that language. From the off we hope to support Odin in Python, Node.js and Golang. These libraries will be able to extract values from the user code and this is generally accepted to be the first stage of the observability pipeline.
Criticality	This is an important aspect of Odin as jobs as defined entirely by the user's code. While a user could write a job without utilising the library, they would not avail of the observability features which can greatly help in times of job failure.
Technical Issues	The issues arising from writing a set of programming libraries comes in the form of how the user will import said libraries. The team must make the libraries available through popular language specific package managers such as pip, npm, and Golan's get functionality which allow imports directly from

	Github.
Dependencies	The libraries we will write as dependent on the existing structure for creating libraries. Each language has a specification for how libraries are defined. Along with this the utilisation of a language library is totally dependant on the success the team has in making it available to language specific package managers.

4. System Architecture

4.1 System Architecture Diagram



The above diagram illustrates how the components of Odin connect to one and other on a single machine or node.

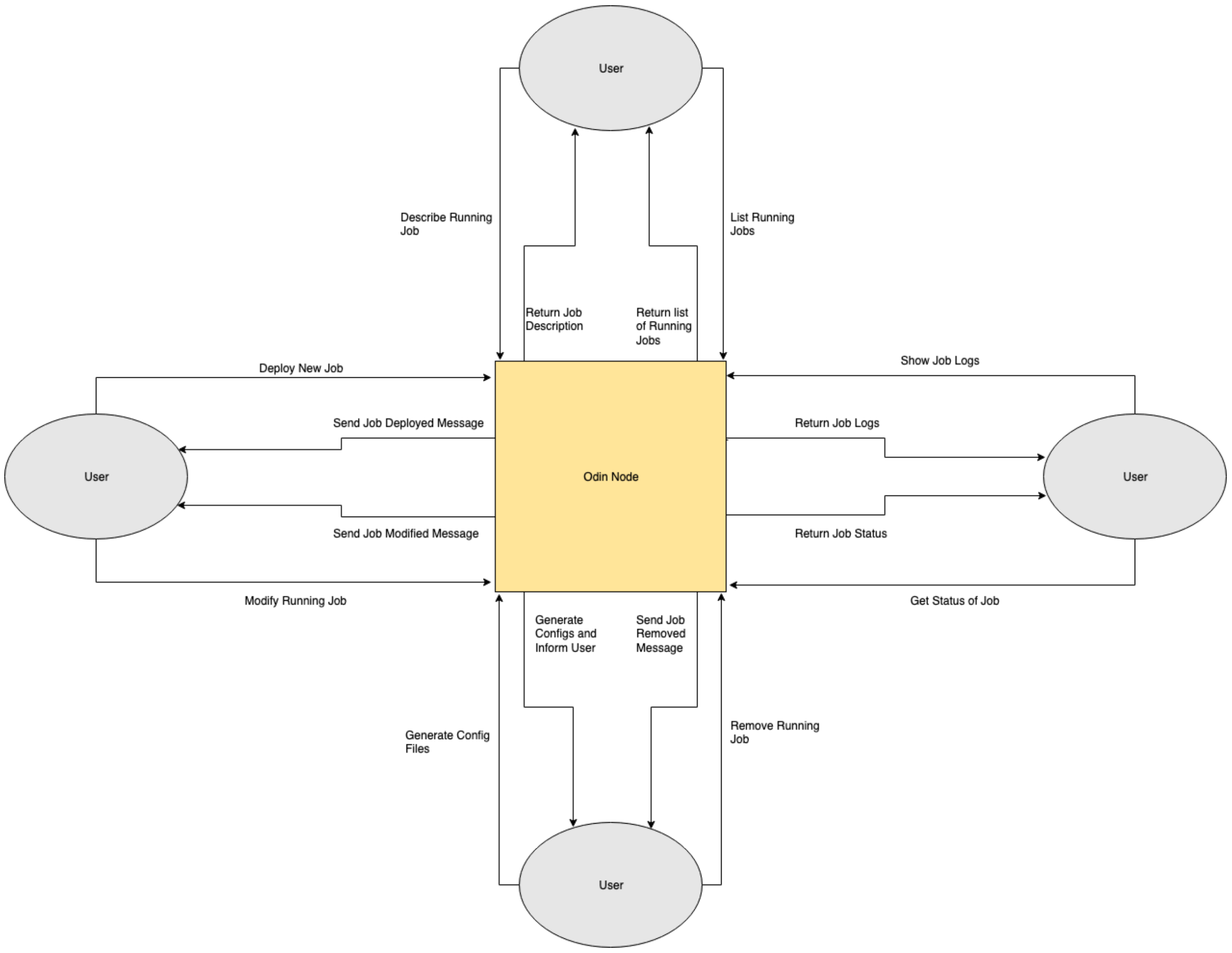
The *odin-cli* component is activated by the user from the command line. Depending on the type of action to be taken (adding, removing, modifying, etc), the *Job Manager* component handles the request. In the case of adding or modifying a job, the *odin-cli* component requires the user code, which imports the *odin-library* component. The latter of these signifies the language specific library Odin imports to observe and monitor code. A job consists of the user code and the respective config file.

Depending on the action being taken, information is passed to either the *Scheduler* (in the case of an add or modify) or straight to the *Priority Queue* (in the case of a removal). In the case of the latter, a job is simply removed from the queue. In the case of the former, metadata extracted from the user code and configuration file is passed to the *Odin Engine*, specifically to either the *Scheduler* component. Once the scheduler has carried out its purpose, the job is then placed in the queue, where it waits to be called upon by the *Executor* component.

The Executor component will execute the job on the machine or node and then pass and values it extracts into a *Time Series Database*. The Web UI will then use information from the *Priority Queue* (for running jobs) and the *Time Series Databases* (recorded values) in order to build an observability platform.

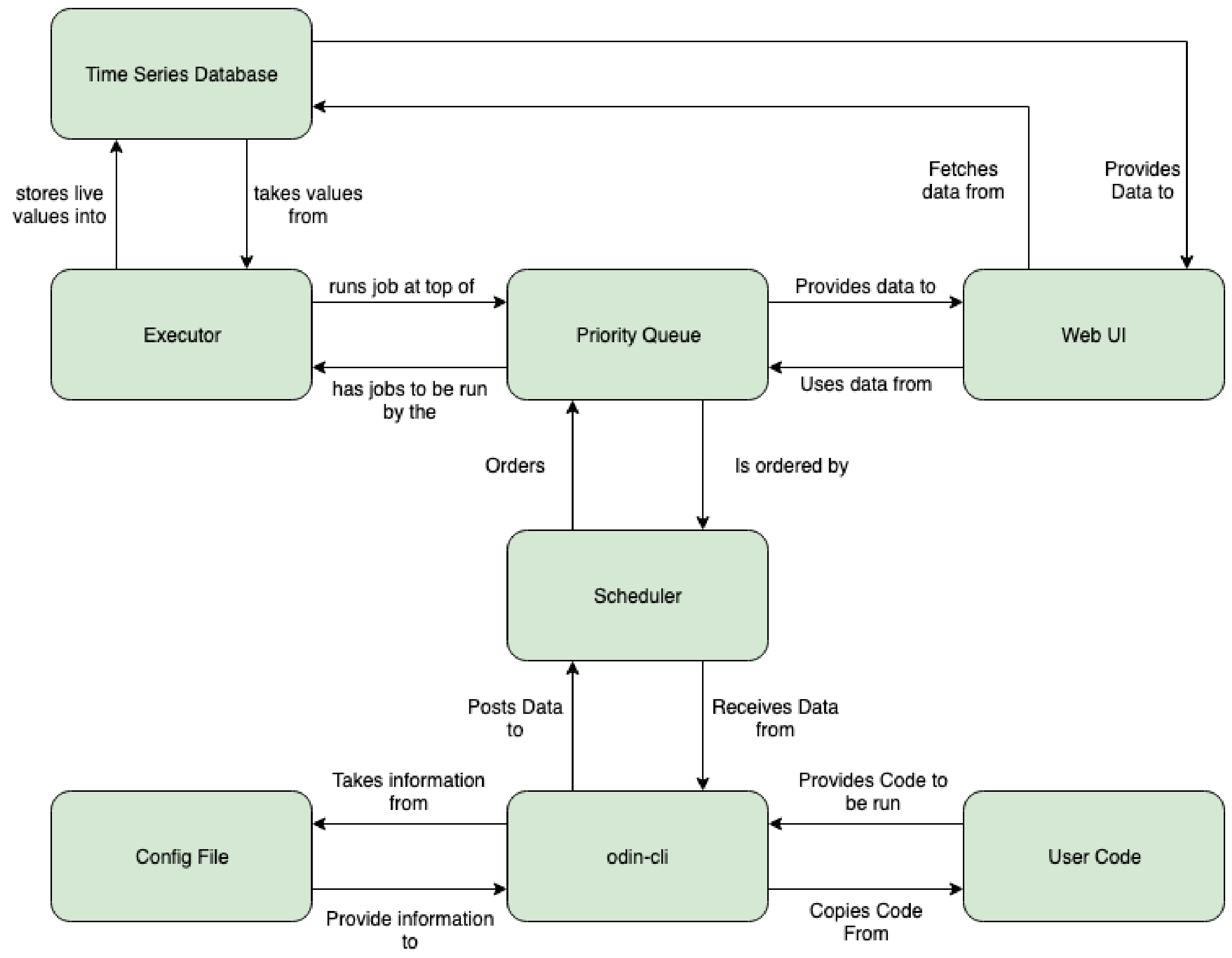
5. High-Level Design

5.1 Context Diagram



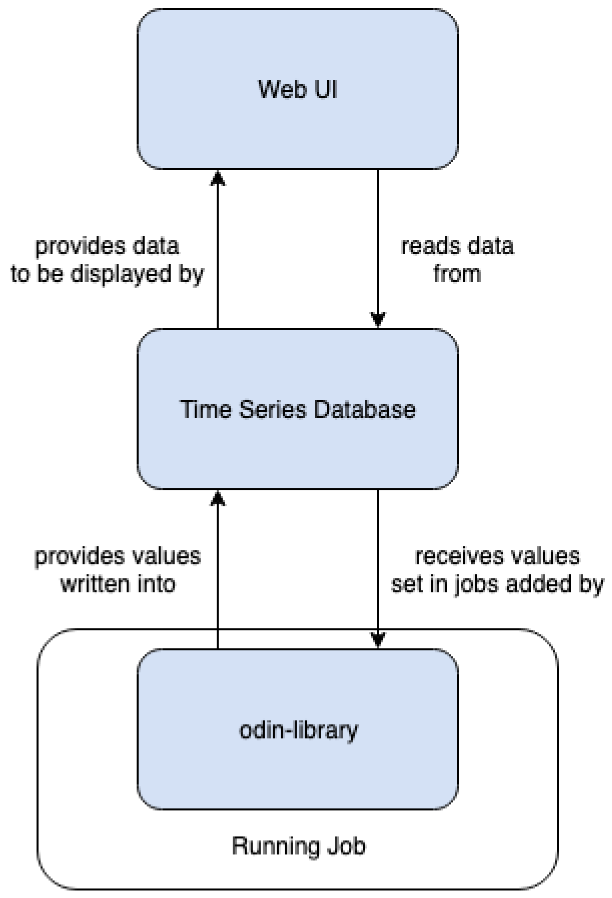
The Context Diagram illustrates how Odin and its surrounding environment interact. This environment consists of users making several different types of queries to the system, from listing jobs, to describing jobs, to removing jobs or showing their logs. For each query sent by a user, the user receives a corresponding response message to confirm the action has taken place successfully.

5.2 Data Flow Diagram - Deploying a Job from the CLI



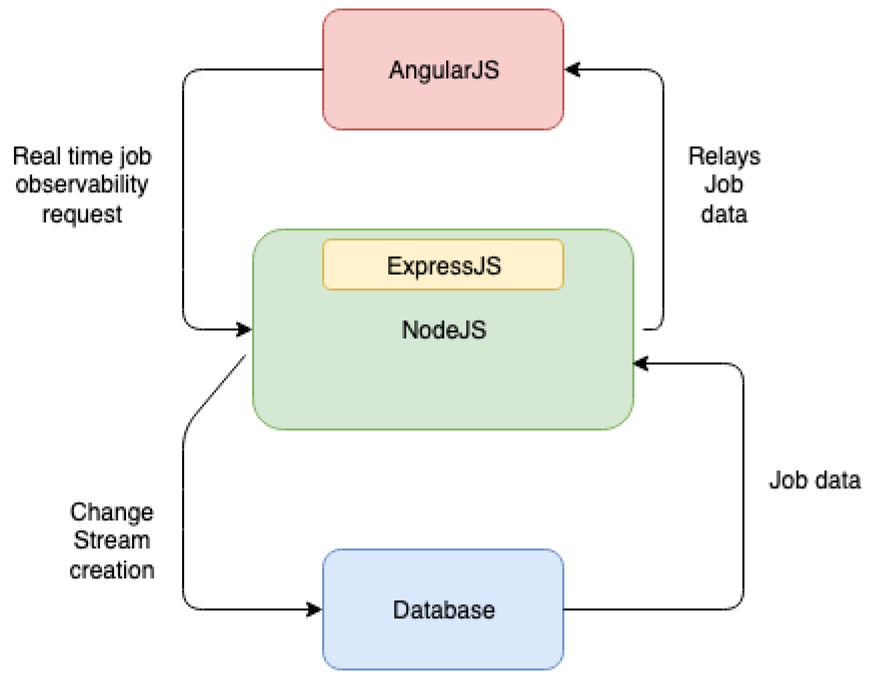
The above Data Flow Diagram details the means by which data flows during the process of deploying a new job. We can see from this diagram that the users code is the true starting point for this process, as it is (along with the corresponding config file) used to build and deploy the job by the *odin-cli*. This job is scheduled and assigned a priority in the queue of jobs. From here when the job is executed the data is provided to the *Web UI* and the *Time Series Database*.

5.3 Data Flow Diagram - The Observability Pipeline


Web UI
provides data to be displayed by
reads data from
Time Series Database
provides values written into
receives values set in jobs added by
odin-library
Running Job

Here the Data Flow Diagram details the way in which data flows through the observability pipeline. We can see the direct link that exists between the language specific libraries written for Odin and the end destination of the *Web UI*. The libraries lend the capture functionality to the system, and when the job is execute it writes values to a *Time Series Database*. From here the data captured may be used to infer the internal state of the job via the results shown on the *Web UI*.

5.4 Data Flow Diagram - Real Time Job Inspection in the Web UI



Here the Data Flow Diagram details how real time data is retrieved from the database and displayed on the Web UI for observability purposes. A user makes a request to see real time data from an executing job. This is triggered by AngularJS sending a request to an ExpressJS API running within Node.js. From here, Node.js creates a change stream with the database. Change streams allow applications to access real-time data changes without directly polling the database, but instead waiting for the database to send out changes. From here the relevant job data is sent back to the ExpressJS API, and this is relayed back to the user through the medium of AngularJS.

6. Preliminary Schedule

Sprint	Start Date	Task to Complete	Person Assigned
1	02/12/19	Scheduler	James
1	02/12/19	Node Library	Martynas
2	16/12/19	Executor	James
2	16/12/19	Web UI	Martynas
-	06/01/20	Exam Period	James
-	06/01/20	Exam Period	Martynas
3	27/01/20	Python Library	James
3	27/01/20	Metrics Collection	Martynas
4	10/02/20	Command Line Tooling	James
4	10/02/20	Golang Library	Martynas
5	24/02/20	Bug Fixes	James
5	24/02/20	Bug Fixes	Martynas
6	16/03/20	Documentation	James
6	16/03/20	Documentation	Martynas
7	30/03/20	Distribution Stretch	James
7	30/03/20	Distribution Stretch	Martynas
8	27/04/20	Bug Fixes	James
8	27/04/20	Bug Fixes	Martynas
-	05/05/20	Exam Period	James
-	05/05/20	Exam Period	Martynas

7. Appendices

1. Distributed Periodic Scheduling - https://landing.google.com/sre/sre-book/chapters/distributed-periodic-scheduling/

2. The Observability Pipeline - https://bravenewgeek.com/the-observability-pipeline/

3. Job Scheduling Techniques for Distributed Systems - https://www.iis.sinica.edu.tw/papers/wuj/9213-F.pdf

4. How to Design a Job Scheduling Algorithm - https://www.cse.huji.ac.il/~feit/parsched/jsspp14/p2-schwiegelshohn.pdf

5. Change Streams in MongoDB - https://docs.mongodb.com/manual/changeStreams/