Odin - Technical Manual

ODIN

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

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

Project Supervisor: Dr. Stephen Blott

Completion Date: 16/05/2020

0. Table of Contents

1. Introduction

1.1 Project Overview

Odin is a programmable distributed 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 run 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 (written in Go, Python and Node.js) 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 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.

Toil - the kind of work tied to running a production service that tends to be manual, repetitive, automatable, tactical, devoid of enduring value, and that scales linearly as a service grows.

MongoDB - a cross-platform document-oriented NoSQL database program which uses JSON-like documents with schema.

SystemD - a Linux service manager that provides a logging daemon and other tools and utilities to help with common system administration tasks.

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.

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.

Raft Consensus Protocol

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.

1.3 Motivation

During the team's respective INTRA placements, both team members dealt with various internally written job/task schedulers and other software such as Jenkins and orchestration tools such as Docker Swarm. We came from teams who aimed to implement the best practices of DevOps and we quickly learnt that DevOps is something to be practiced in the tools a team uses.

When postulating ideas for our final year project, we shared our experiences with such technologies from our INTRA placements, and we began to note the absence of a true open source alternative to job orchestration that was not built on top of existing orchestration technologies like Docker Swarm, Kubernetes or Apache Mesos.

Odin is useful as it gives teams a shared platform for jobs that allows individual members to package their code for periodic execution. Simply, users turn their code into scheduled individual jobs, or indeed, job chains. Specifically, Odin would be of benefit to teams developing on Linux machines.

Odin is useful to users and teams from an observability point of view, providing a set of metrics and variable insights which will in turn lend transparency and understanding into the execution of all system run jobs.

1.4 Research

Before spending time on initial design implementation details, we acknowledged the importance of prior research. As we aimed to build a distributed scheduling tool for teams practicing DevOps we consulted the following chapter in the now famed Google SRE Handbook:

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

This chapter specifically details two approaches to scheduling managers, those that store state and those that use remote data stores to hold information. The chapter details the advantages and disadvantages of both, with specific emphasis on the dangers of relying on a single source dependency such as a remote data store. From our reading of this chapter we set sights to ensure that Odin, which would utilise a remote data store, would be able to manage its own execution state in the event the data store went offline.

We understood the importance that observability would play in the Odin ecosystem. The team consulted with the following two appendices in particular:

- The Observability Pipeline - https://bravenewgeek.com/the-observability-pipeline/
- Charity Majors Observability Blog - https://charity.wtf/tag/observability/

The former of these writings really gave us a direct insight into how observability systems work, with direct data collectors/loggers which run information through concurrent pipelines into the data stores. This confirmed to us that the correct approach in designing our system was for the language specific software development kits to extract live information about jobs, and then plugging this into a backend of a visualisation unit.

The latter of these writings, by observability thought leader Charity Majors, focused on the cultural aspect of observability within teams, and this writing really focused on how observability is a prerequisite for any major effort to have saner systems. Observability allows built-in transparency to the user in the instance of Odin, so these writings allowed us to broaden our scope in regards to what observability in a system could help us achieve.

Finally we consulted the following two appendices in relation to the algorithmic side of job scheduling, along with performance techniques to enhance the rate of execution:

- Job Scheduling Techniques for Distributed Systems - https://www.iis.sinica.edu.tw/papers/wuj/9213-F.pdf
- How to Design a Job Scheduling Algorithm - https://www.cse.huji.ac.il/~feit/parsched/jsspp14/p2-schwiegelshohn.pdf

2. System Architecture

2.1 Programming Language and Build Tools

2.1.1 Programming Languages

- Go
  - Version: 1.13
  - Usage: Odin Engine, Odin CLI, Odin Go SDK
- Python
  - Version: 3.7
  - Usage: Odin Python SDK
- Node.js - TYPESCRIPT TOO????
  - Version:
  - Usage: Odin Dashboard, Odin Node.js SDK

2.1.2 Build Tools

- GNU Make
  - Version: 4.1
  - Usage: Automating the local building and testing of the Odin Engine and Odin CLI

2.2 Dependencies

2.2.1 Odin Engine

- Chi
  - Version: v4.0.4
  - Description: A lightweight, idiomatic and composable router for building Go HTTP services.
  - Source Code: github.com/go-chi/chi
  - Usage: Scaffolding the central API for the scheduling and execution of jobs.

- Cronexpr

    - Version: v0.0.0
    - Description: A Golang cron expression parser.
    - Source Code: github.com/gorhill/cronexpr
    - Usage: Parsing cron schedule strings into equivalent timestamps.

- Raft
    - Version: v1.1.2
    - Description: A Golang implementation of the Raft consensus protocol.
    - Source Code: github.com/hashicorp/raft
    - Usage: Managing a replicated log which is used with a finite state machine to manage replicated state machines.

- Logrus
    - Version: v1.5.0
    - Description: A structured and pluggable logging library for Go projects.
    - Source Code: github.com/sirupsen/logrus
    - Usage: Maintaining a consistent format for job execution logs.

- MongoDB Go Driver
    - Version: v1.3.1
    - Description: The MongoDB supported driver for Go.
    - Source Code: github.com/mongodb/mongo-go-driver
    - Usage: Interacting with MongoDB to store and access job information and any associated metrics.

- Go Yaml
    - Version: v2.2.8
    - Description: A package that enables YAML support for Go.
    - Source Code: github.com/go-yaml/yaml
    - Usage: Encoding to and decoding from YAML configurations used for the setup of the Odin Engine.

2.2.2 Odin CLI

- Cobra
    - Version: v0.0.6
    - Description: A library for creating powerful modern CLI applications.
    - Source Code: github.com/spf13/cobra
    - Usage: Scaffolding the command line tool subcommands and flags used to interact with the Odin Engine.

- Go Yaml
    - Version: v2.2.8
    - Description: A package that enables YAML support for Go.
    - Source Code: github.com/go-yaml/yaml
    - Usage: Encoding to and decoding from YAML file used to configure jobs.

2.2.3 Odin Software Development Kits

- Go
    - MongoDB Go Driver
        - Version: v1.3.1
        - Description: The MongoDB supported driver for Go.
        - Source Code: github.com/mongodb/mongo-go-driver
        - Usage: Interacting with MongoDB to store import information from your code into the MongoDB observability collection.

    - Go Yaml
        - Version: v2.2.8
        - Description: A package that enables YAML support for Go.
        - Source Code: github.com/go-yaml/yaml

        - Usage: Handling YAML passed from the job code.

- Python
    - Pymongo
        - Version: v3.10.1
        - Description: The MongoDB supported driver for Python.
        - Source Code: github.com/mongodb/mongo-python-driver
        - Usage: Interacting with MongoDB to store import information from your code into the MongoDB observability collection.
    - Ruamel.yaml
        - Version v0.15
        - Description: A library that enables YAML support for Python.
        - Source Code: sourceforge.net/projects/ruamel-yaml/
        - Usage: Handling Yaml passed from the job code.
- Node.js
    - Mongodb
        - Version ^3.5.6
        - Description: The official MongoDB driver for Node.js. Provides a high-level API on top of mongodb-core that is meant for end users.
        - Source Code: https://github.com/mongodb/node-mongodb-native
        - Usage: Interacting with MongoDB to store import information from your code into the MongoDB observability collection.
    - Yamljs
        - Version ^0.3.0
        - Description: Standalone JavaScript YAML 1.2 Parser & Encoder. Works under node.js and all major browsers. Also brings command line YAML/JSON conversion tools.
        - Source Code: https://github.com/mongodb/node-mongodb-native
        - Usage: Handling Yaml passed from the job code.

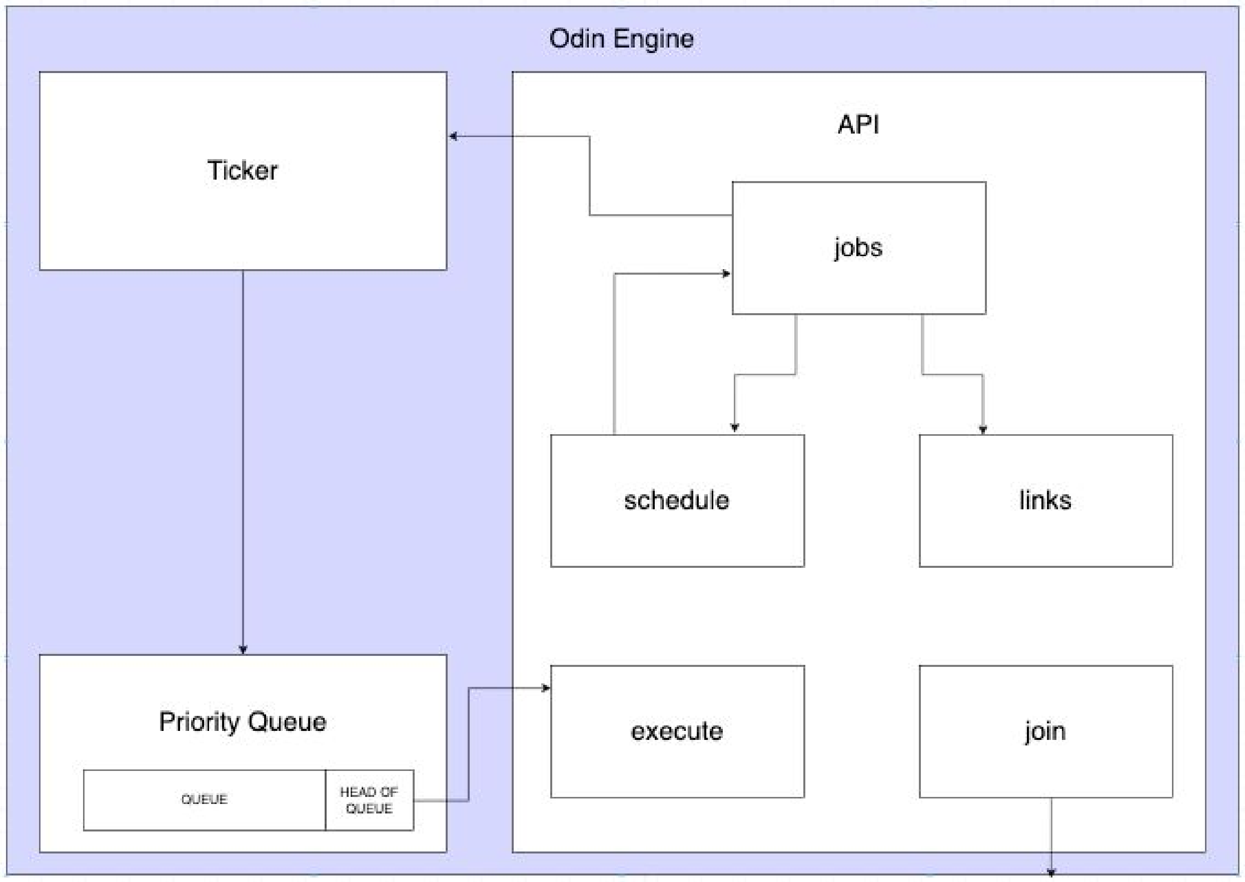
2.2.4 Odin Dashboard

- Node.js

  Due to an extensive list of dependencies typical for web applications, the dependencies for the Odin Observability Dashboard can be found here:

  - src/webapp/client/package.json
  - src/webapp/server/package.json

2.3 The Odin Engine



Above is a high level overview of the data moves internal to the Odin Engine. We can see it is made up of three main components, the API, the Ticker and the Priority Queue. Any external data passed into this system runs through the API.

2.3.1 API

The backbone of the Odin Engine is the API, built using the Go-Chi framework. The API has five endpoints the user leverages to deploy, schedule and maintain jobs:

- Jobs

  The Jobs endpoint is where a great deal of external data is passed into the system. This endpoint is focused on interacting with other internal endpoints such as Schedule and Links. This is because the main purpose of the Jobs endpoint is to interface and manipulate job data in MongoDB.

- Schedule

  The Schedule endpoint is tasked with parsing the Odin Schedule String found in a jobs YAML config. The Odin Schedule String format is an abstraction over the Cron Schedule String format, so this endpoint deconstructs the provided YAML into a Cron Schedule String.

- Links

  The Links endpoint is used to manage links between jobs. A link between two jobs ensures that when the first job has successfully executed, the second job is called to be executed. The main operations here included linking and unlinking jobs from one and other.

- Execute

  The Execute endpoint is tasked with performing execution on the head of the priority queue. This endpoint will use the orchestrated path of the user code and execute it with the UID and GID of the user who deployed it.

- Join

  The Join endpoint is used to add nodes to the Raft cluster, which runs by default on port 12000. Recommendations for distributed architecture can be found in section 4.5 of the Odin User Manual

As with systems like Docker, access to the central API in the Odin Engine is only granted if the user is a member of the odin group on the Linux system. This is a security measure to ensure access is controlled.

By default, parameters such as the address and port of the API are set in the odin-config.yml file found in the root user's home directory. Along with this exist the `ODIN_MONGODB` and `ODIN_EXEC_ENV` environment variables, which are used to

2.3.2 Ticker

The ticker is used to track the time until execution of each job in the Priority Queue. The queue is therefore polled regularly to see if it is time to run a job. When a job is found to be at it's time of execution, the job is passed from the queue to the execute endpoint of the API.

Any jobs executed are then passed back to the schedule endpoint so as they are scheduled back into the priority queue.

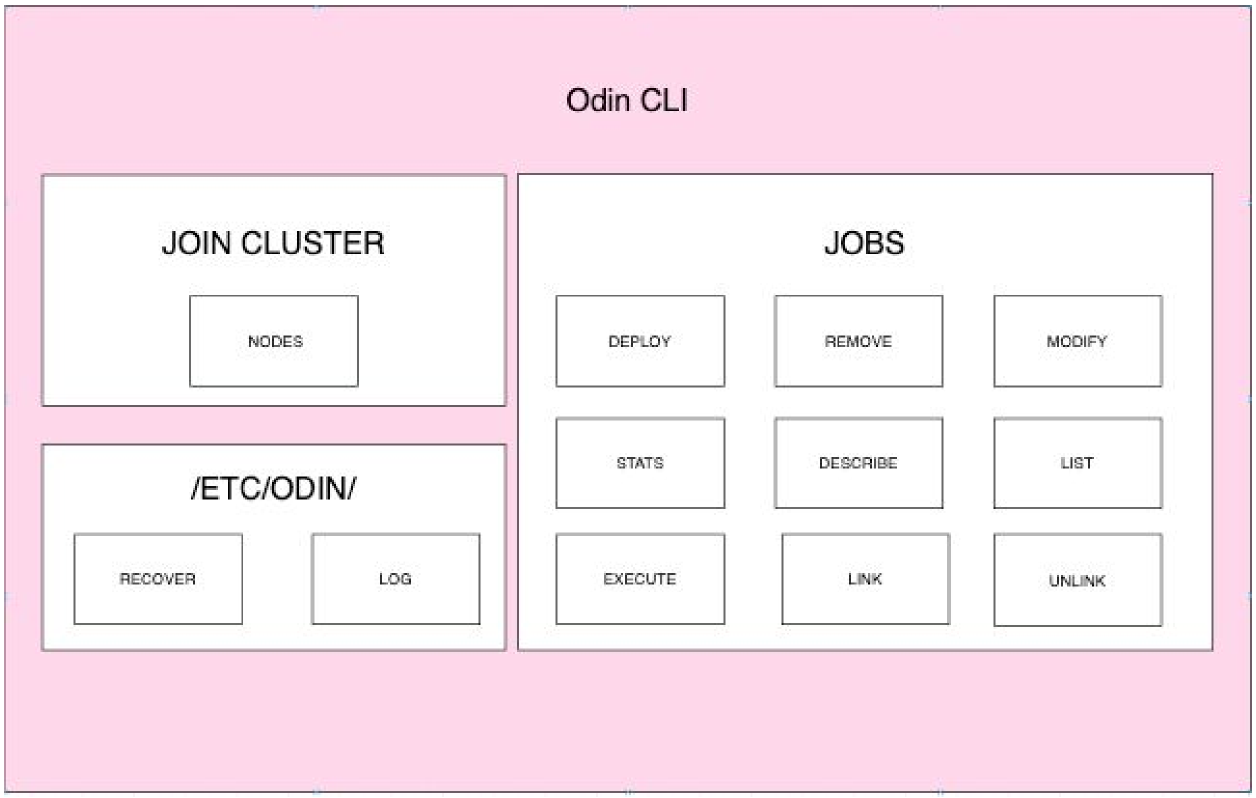
2.3.3 Priority Queue

The Priority Queue is where the order jobs are kept in wait until it is time for execution. Priority in this structure is directly influenced by the time until execution.

Jobs are ordered from those with the least time until execution to those with the most time until execution. Often multiple jobs will be scheduled to execute at the same time. Due to this, instead of each node in the queue representing a job, for efficiency we have made each node in the queue represent a map of an integer (seconds until execution) to an array of jobs which will execute at this time.

Each job in the array has a field pointing to the user code to be executed once the time of execution has arrived. This code is found in the `/etc/odin/jobs` directory under a directory of that job's ID.

2.4 The Odin CLI



The Odin CLI acts as the entrypoint to the Odin Engine. The CLI is responsible for providing users an easy way to deploy jobs, remove jobs, and maintain and interact with running jobs. Along with this the CLI allows users to expand the current Raft cluster.

In the diagram above, we see the three categories of operation under which Odin CLI commands fall:

- Jobs
- Join Cluster
- /etc/odin

2.4.1 Jobs

A great deal of the commands in the Odin CLI fall under this category. These commands are used to interface with the jobs endpoint in the Odin Engine, which in turn can interface with the MongoDB instance, the Ticker, and the schedule and link endpoints.

These operations usually require parameters such as files to deploy or job ID's to lookup.

2.4.2 Join Cluster

The `nodes` command is the only one to fall into this category. This is specifically used to expand the size of the current Raft cluster by adding a new Odin Engine node.

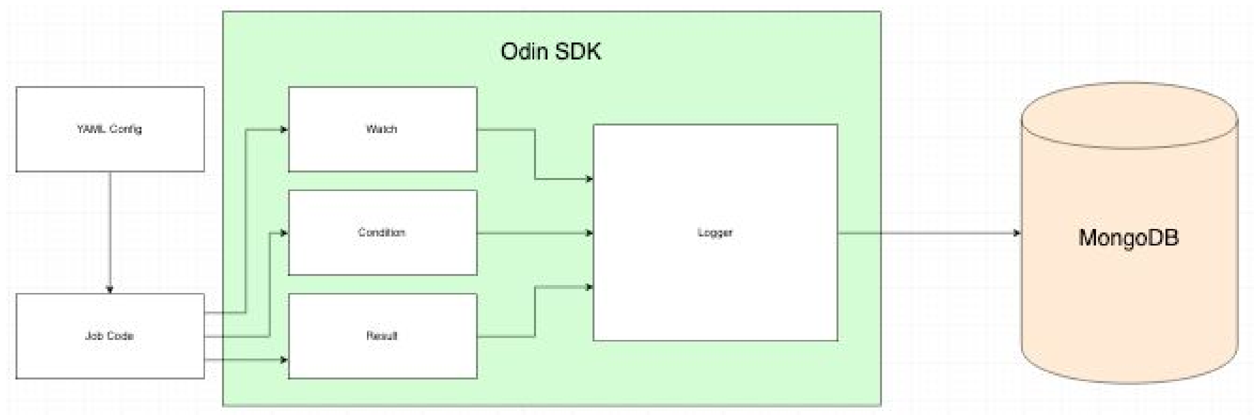
2.4.3 /etc/odin

The two commands to fall under this category are `recover` and `log`. These commands directly communicate with files found under `/etc/odin/jobs` and `/etc/odin/logs` respectively.

In the case of the log command, the CLI will read a specific jobs log file and return it to the user. In the case of the recover command, the CLI will fetch the YAML and user code for a specific job from the /etc/odin/jobs directory.

2.5 The Odin Software Development Kits

Despite the different runtimes, the Go, Python and Node.js software development kits are, architecturally speaking, similar.

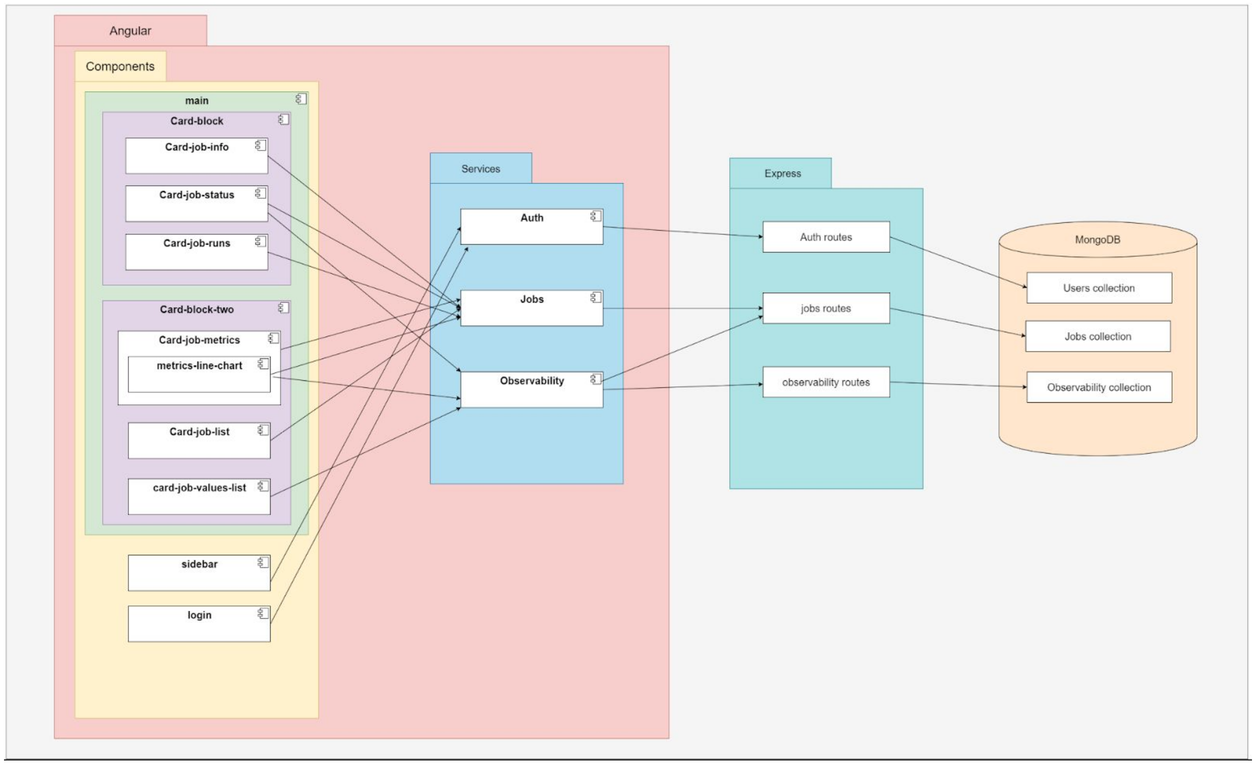


The job code will fetch some metadata from the Yaml config, after which one of three functions can be called on the object created. These functions are Result, Condition, and Watch. In regards to the workings of each function, please consult section 4.3 of the Odin User Manual.

All three of these functions are fed into a singular logging instance which interfaces directly with the MongoDB instance. This is the process by which values from user jobs are observed, with each value logged alongside a timestamp.

2.6 The Odin Dashboard

Dashboard Data Flow

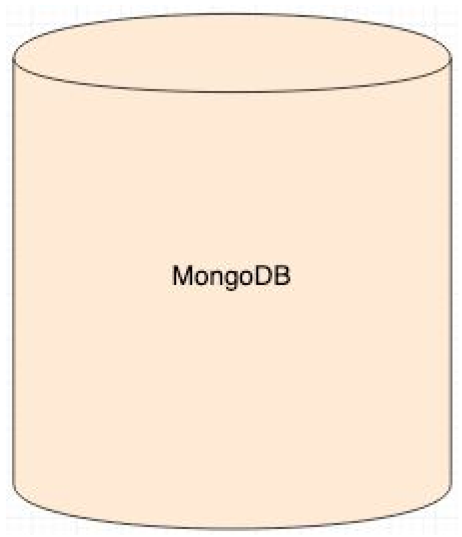


The Odin observability dashboard has three main parts: front end, back end and database . TheAngular front end consists of components which use services to communicate with NodeJS express backend server which interacts with data from our shared mongoDB. The reason why we chose MongoDB is covered in 2.7 MongoDB.
Observability data is updated in near real time by efficient data processing, fast Odin SDKs and the speed that comes with MongoDB.

We aimed Odin to be suitable for use by organisations and that’s why we ensured that their data is protected.

Our auth system supports OAuth2 for google sign in and due to modular approach it could be easily expanded to support more OAuth2 login providers if an organisation has a different preference. All backend routes are protected with Json web token verification to ensure that the token was created using our secret keys while front end components are protected by the auth service guards.

2.7 MongoDB



A central part of this system is the MongoDB instance. Be this a local installation or something provisioned through a cloud provider, the data store of Odin is integral to its operation.

We acknowledged early on that one of the most important aspects of our system was execution. If at any point MongoDB went offline, jobs would not execute as execution was dependent on the availability of the data store. After noticing this, we decided that the execution centre should hold some state too, in the case it does go offline.

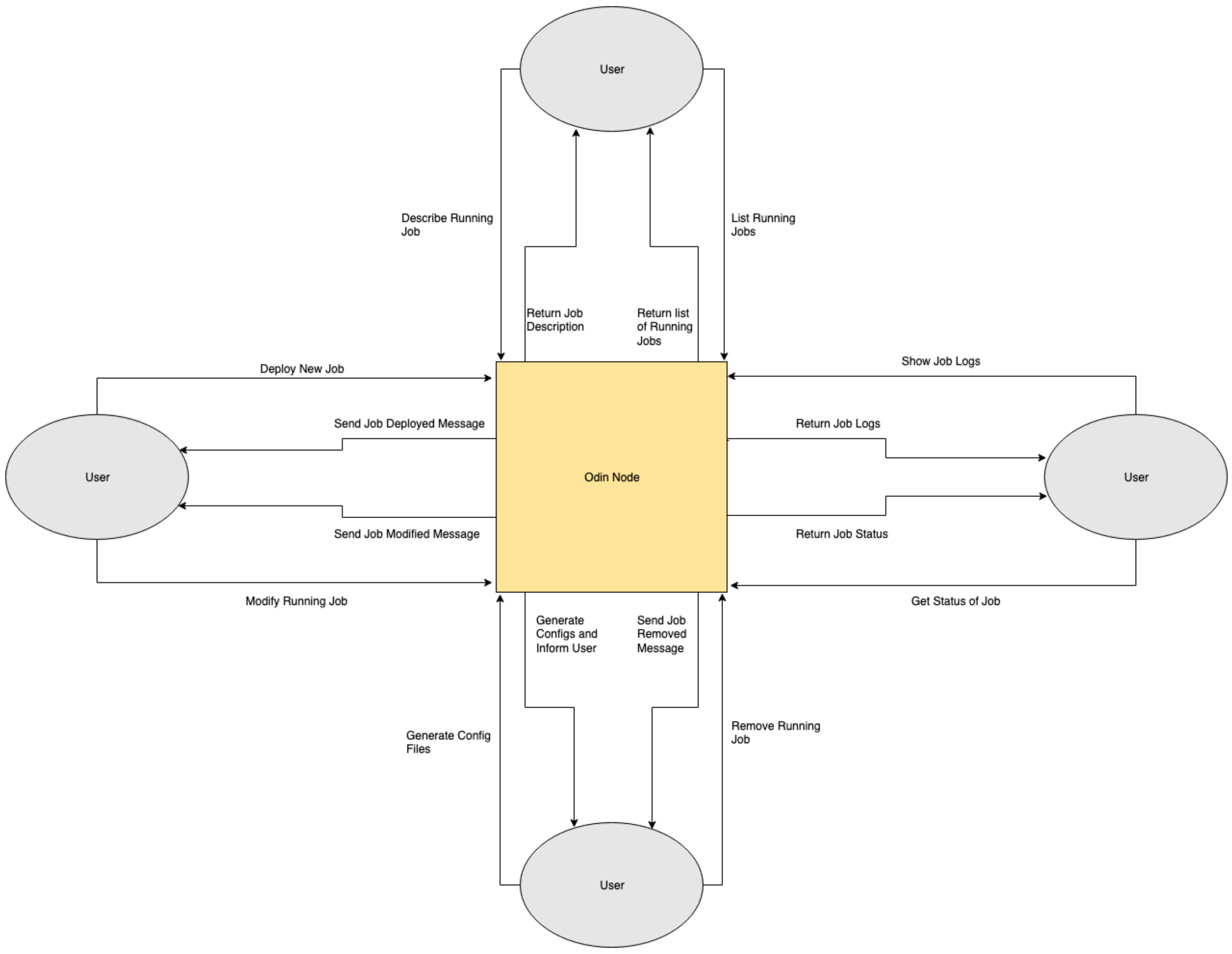
This redundancy now means that if the system cannot connect to MongoDB it will in no way affect the execution process. While in this state, jobs cannot be removed, added, or modified. This is because our system wishes to preserve the integrity of the data store when it went offline. Any jobs which are in the priority queue will execute without fail.

3. High-Level Design

3.1 Initial Design

In the beginning the team focused on Context Diagrams (to illustrate how Odin and its surrounding environment interact) and Data Flow Diagrams (to detail the means by which data flows during a given process).

3.1.1 Context Diagram

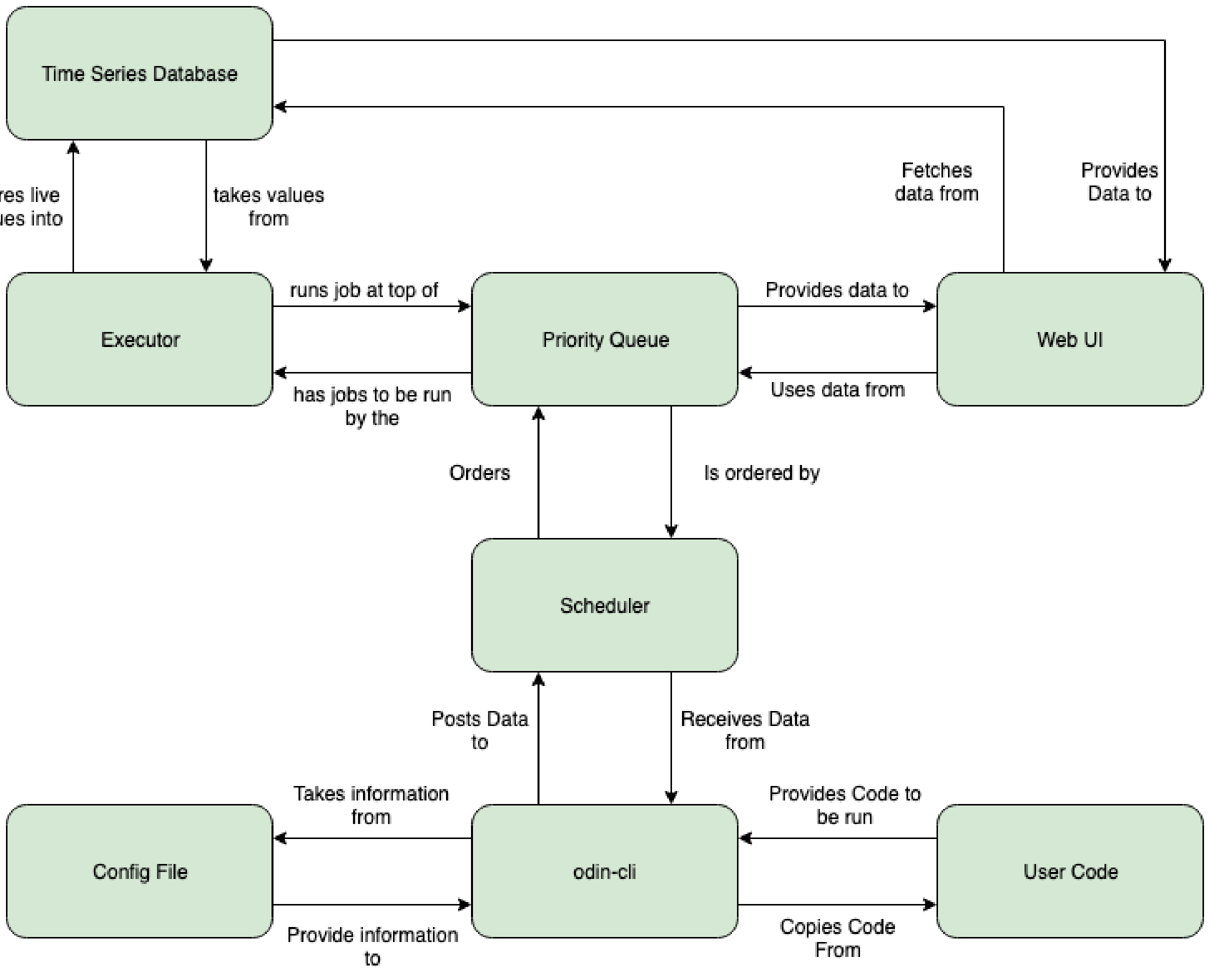


As an initial design, this context diagram reflects a significant amount of functionality found in Odin CLI commands. The user will still directly make requests to the central engine, and will receive an appropriate response.

The only functionality found in our initial design which we did not keep was the ability to get the status of a job. We removed this as the `odin log` command can already be used to check the status of execution. We removed this status functionality and opted for an `odin stats` command instead, which displays the stored values from user code.

This context diagram is also lacking in how we would deal with error messages. Along with this, the diagram fails to lend any context to specifically how the "Odin Node" deals with information in a way which facilitates the user with a response.

3.1.2 Data Flow Diagram - Deploying a Job from the CLI



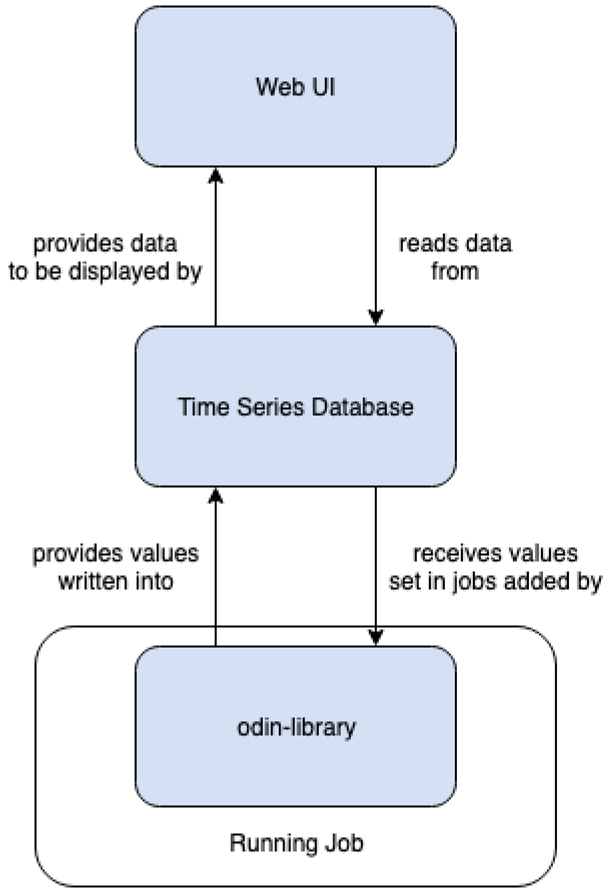
This Data Flow Diagram is still accurate in some regards. The Odin Cli still fetches data from the YAML configuration file and the User Code and will post this data to the scheduler which in turns is used to order the Priority Queue. The Executor will then execute on this queue and store information in a database.

The first real deviation we see is that the links between the Priority Queue and the Web UI (Dashboards) no longer exist. Instead, an Angular backend connects directly to the data store.

Originally we aimed to use a time series database, but found that a NoSQL alternative like MongoDB was more applicable for our use case. It reduced the amount of overhead complexity in the storage side of the project.

The design was lacking as it did not concern itself with how the scheduler and executor components would link up outside of accessing the Priority Queue.

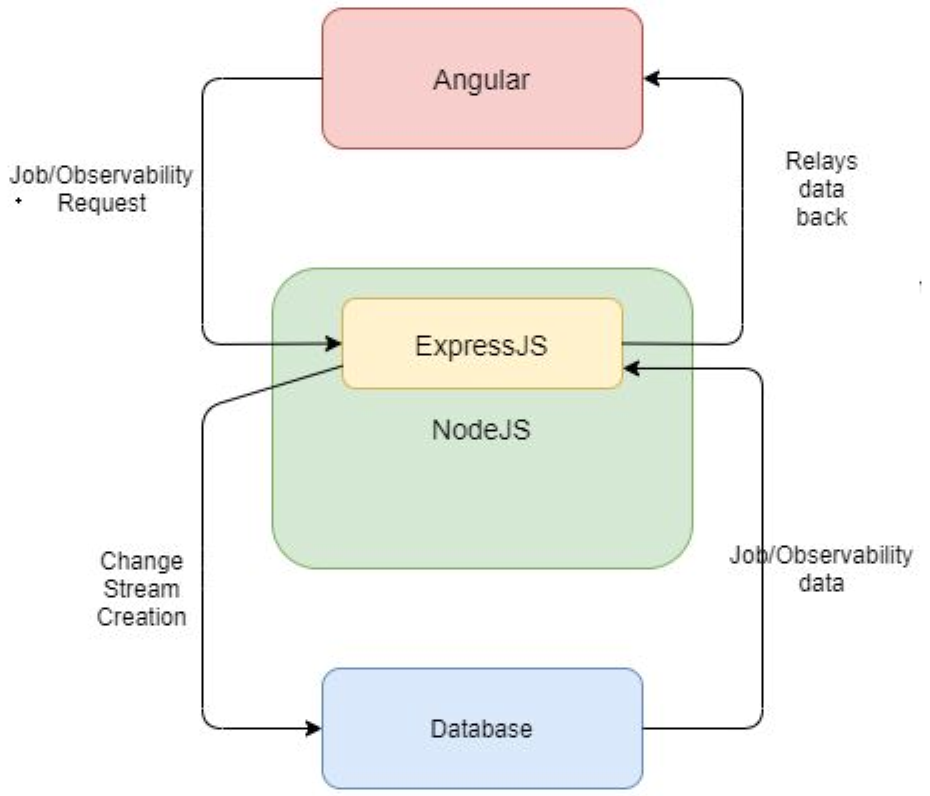
3.1.3 Data Flow Diagram - The Observability Pipeline


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

This Data Flow Diagram is still accurate in most regards, with the greatest deviance from the original design coming from opting for MongoDB over a time series database, as explained in section 3.1.2 of this document.

While operations are more varied in the final design, this stil aptly shows how data flows through the pipeline.

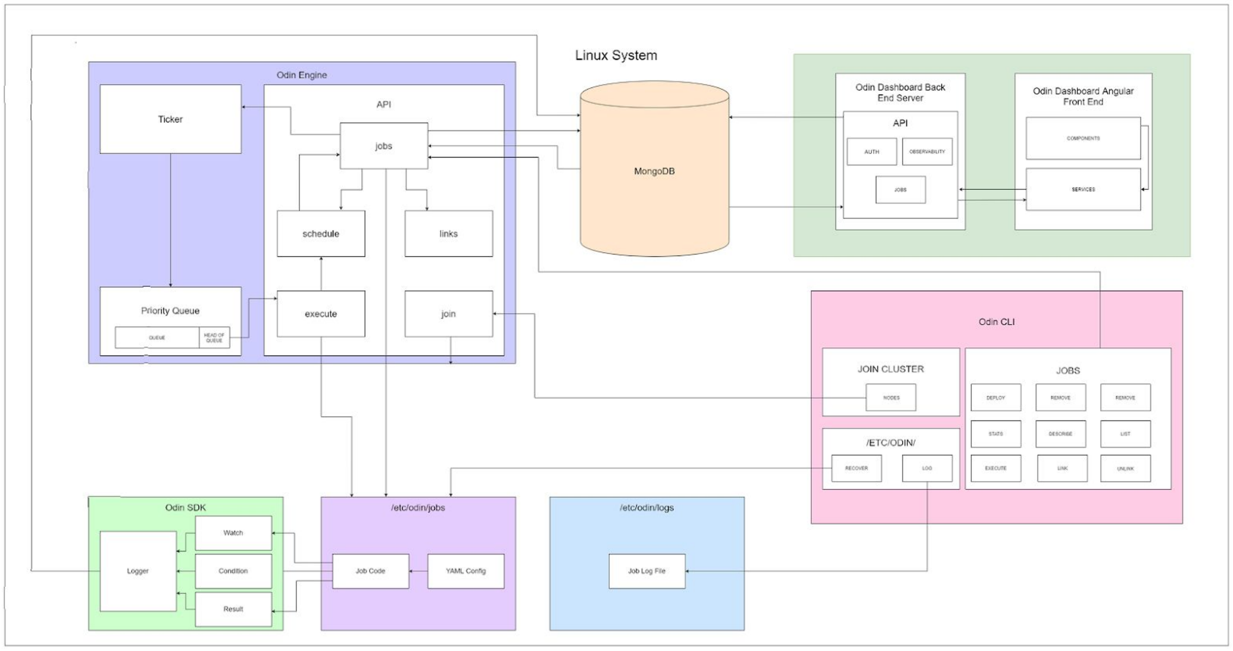
3.1.4 Data Flow Diagram - Job Inspection in the Web UI



While the data flow is similar in final design, change streams are not used and are replaced with regular database queries as change streams required a more complex implementation which we found unnecessary.

3.2 Current Design

3.2.1 System Architecture Diagram



The above diagram represents Odin as it currently operates, making reference to the components (and their respective diagrams) found in sections 2.3, 2.4, 2.5, 2.6 and 2.6 of this document. This diagram serves as a demonstration of how these components link up to facilitate the flow of data. This architecture diagram goes to great lengths to show the internals of each component in a more structured way, something which early design diagrams did not include.

Beginning at the Odin CLI, we can see how each of the three aforementioned categories will link to three different places, with the job commands category linking to the jobs endpoint of the API, and the Join Cluster category linking to the join endpoint of the API. Finally we see how the recover and log commands in the /etc/odin/ category link to their respective sections on the filesystem.

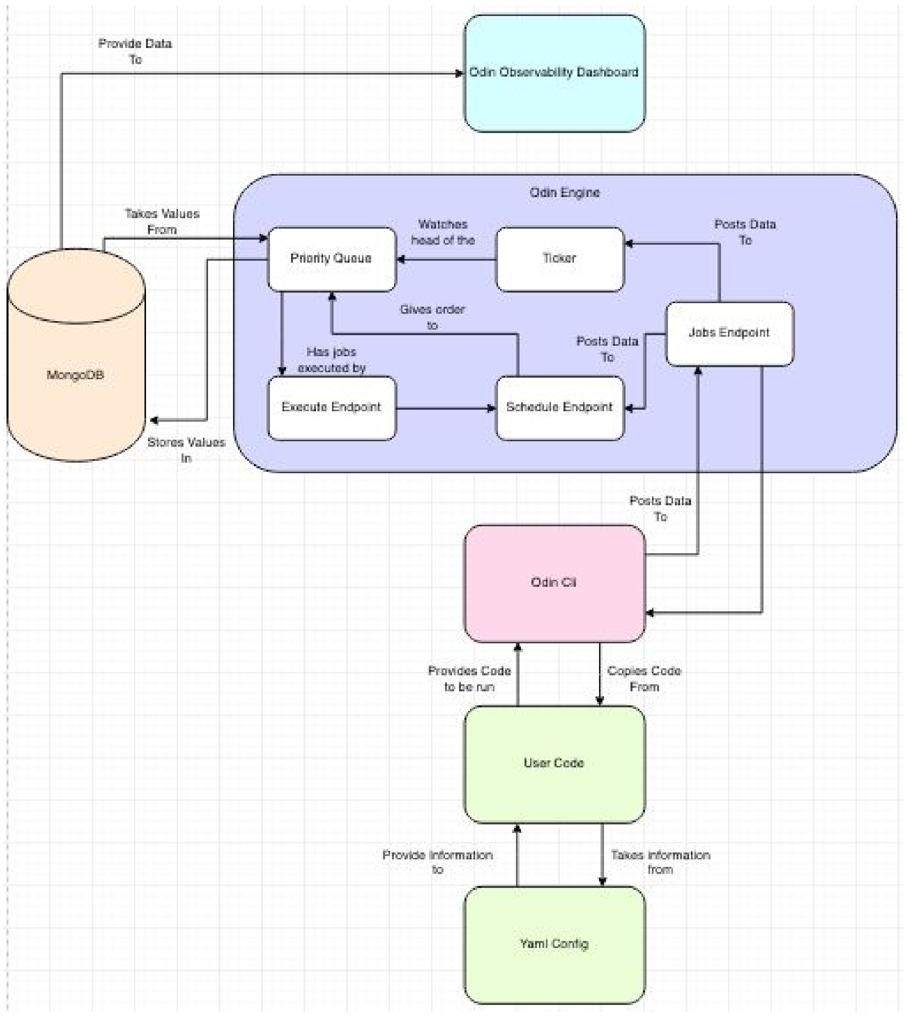
We also see how the jobs endpoint itself is integral to the Odin Engine component. It interacts directly with the MongoDB instance, the Ticker, and the execute and links endpoints. The latter of these, the links endpoint, is something in the current design which was not present in the initial system design. The concept of linking jobs together was discussed in our Function Specification document but never truly explored. We can see this concept has been manifested through the presence in the Odin Engine through the links endpoint.

The join endpoint simply points back to the Odin Engine component itself as a way to signify the replication of the service in a Raft cluster. The Odin CLI directly links to this endpoint to facilitate the ease of expanding this cluster.

The user code which is actually executed is always stored in /etc/odin/jobs under a directory of the job’s ID value. This was a design detail which was never explored in the original design - how orchestration would be achieved. Making a copy of the files used to run a job ensures redundancy, as users may accidentally remove files scattered across their home directory.

We can see how the Odin SDK (of any language) is used to log values and report them to the MongoDB instance. These values can be extracted by the Odin Dashboard backend server sent to be displayed on an Angular frontend.

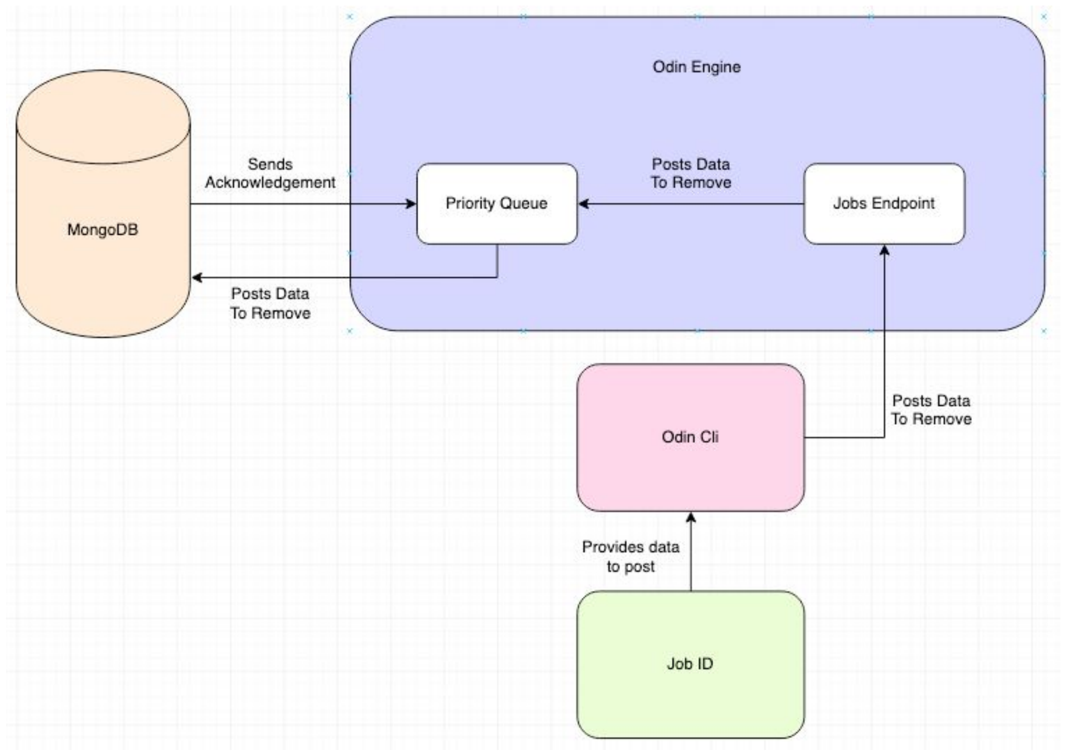
3.2.2 Data Flow Diagram - Deploy a Job



We can see in the above DFD the process of deploying a job touches on most of the components which make up the Odin system.

The auto generated files are taken by the CLI, passed through various endpoints of the Odin Engine, stored in the MongoDB instance and finally displayed on the Observability Dashboard thereafter.

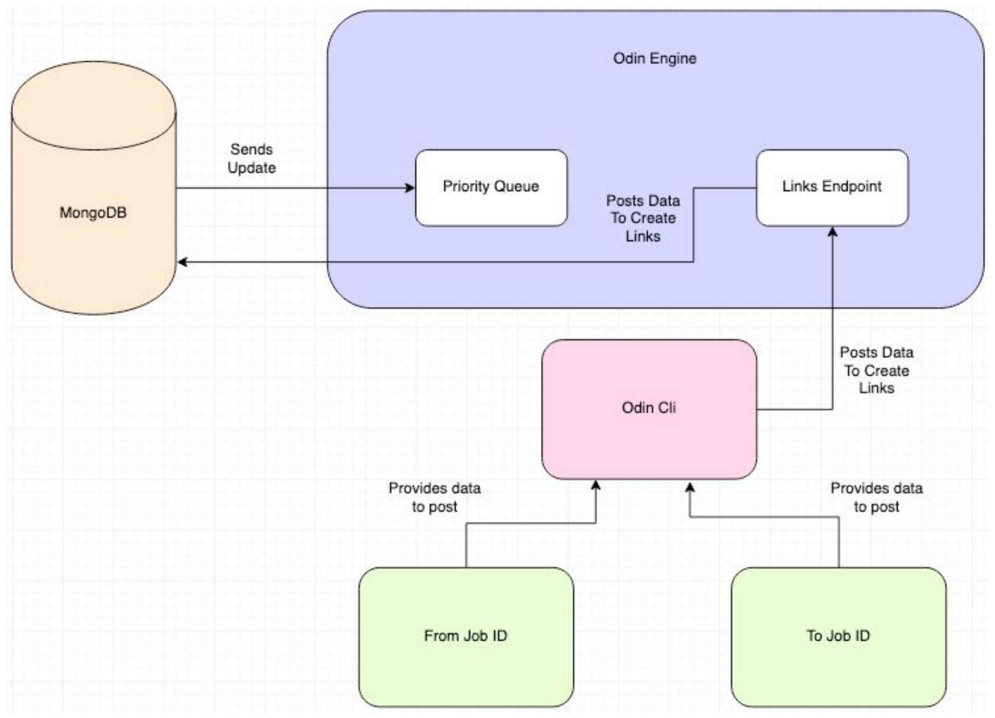
3.2.3 Data Flow Diagram - Removing a Job



The above DFD details the process of removing a job in the Odin system. This is not as contrived as adding a job, as the CLI simply posts the ID of the job to be removed to the Odin Engine.

The Engine will interface with the MongoDB instance to remove this item from the jobs collection and this information will be relayed to the Priority Queue.

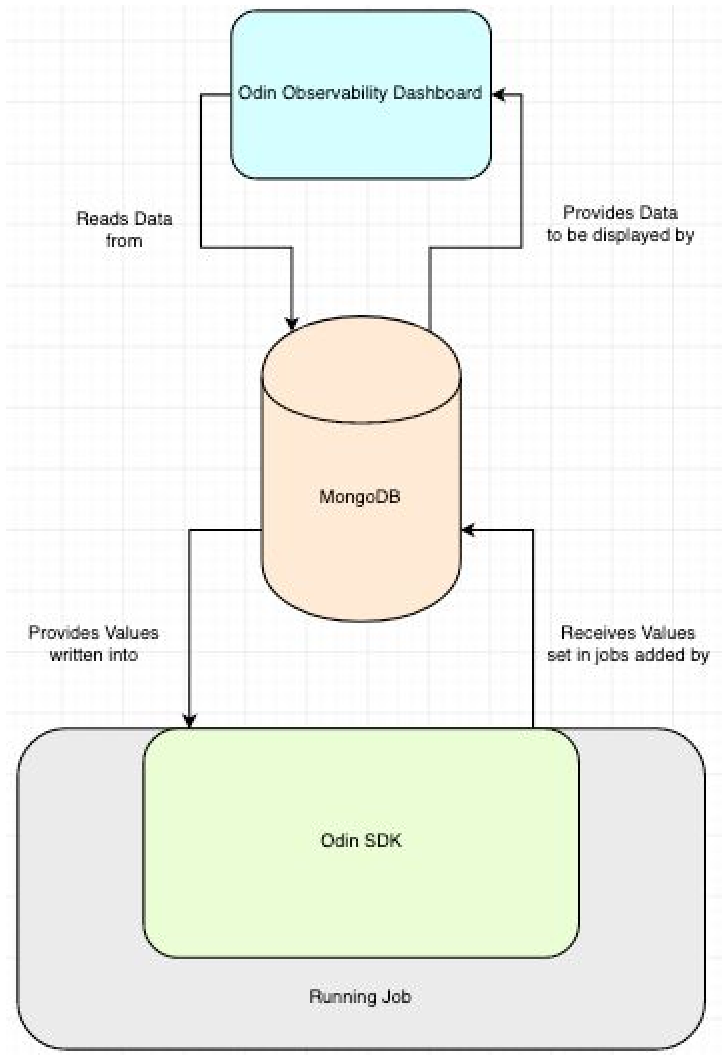
3.2.4 Data Flow Diagram - Linking Jobs



The process of linking jobs requires two jobs to already exist. Each of these jobs has a unique ID, both of which are passed to the CLI.

The CLI posts these ID’s to the Links endpoint and the link field of the “from” ID is updated to point to the “to” ID. This change to the jobs collection is then sent to the Priority Queue so that links may be executed after the calling job is executed.

3.2.5 Data Flow Diagram - The Observability Pipeline


Odin Observability Dashboard
Reads Data from
Provides Data to be displayed by
MongoDB
Provides Values written into
Receives Values set in jobs added by
Odin SDK
Running Job

The observability pipeline, as previously stated, differs very little from the pipeline set out in our original designs. The only difference is the swap of a time series database in favour of an instance of MongoDB. This design decision was made as MongoDB is more applicable for our use case. It reduced the amount of overhead complexity in the storage side of the project.

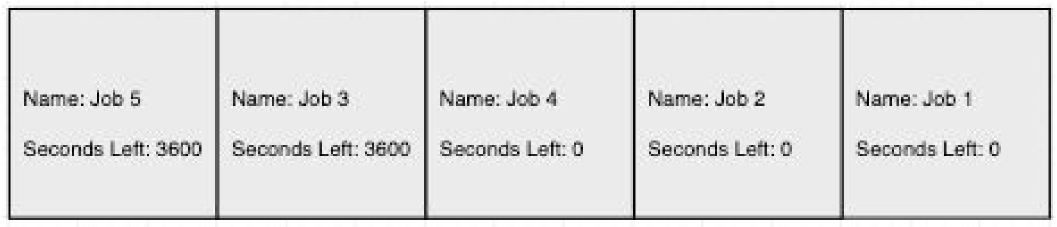
The running job will utilise the language specific Odin SDK to store values in MongoDB, which in turn will be used to generate metrics in the Odin Observability Dashboard.

4. Issues

4.1 The Job Queue

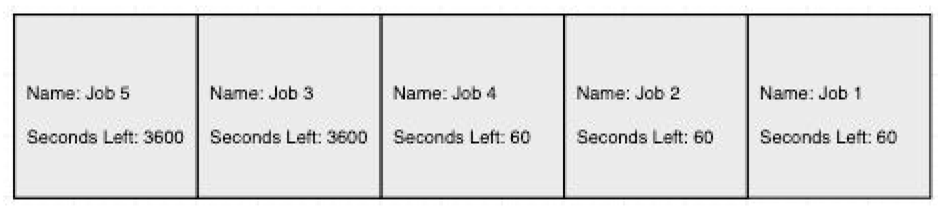
Originally the jobs queue was a simple array of nodes, with each node representing a job. One attribute of each node was that node's time until execution. When the time left was found to be zero, the job would be sent to the executor and the ticker would check the next job in the queue.

The issue with this design was that at any one time, there could be several nodes that had "zero" seconds until execution. This would mean that by the time one job had been sent to the executor, the next job had been rescheduled as it had strayed into a negative range. The diagram below demonstrates the and example of such a queue:

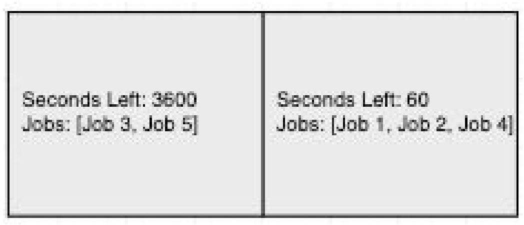


Job 4, Job 2 and Job 1 would all be set to execute at the same time, but only Job 1 would execute successfully, as it was the first job added of these three. The same issue will rise its head again when both Job 3 and Job 5 come to a Seconds Left value of 0 also.

The solution to this problem was simple - groups. Instead of one job per node, we instead tried one Seconds Left value per node. This means that nodes in the queue were not representative of just one job, but rather of all jobs which were set to execute at the same time. This approach meant that this example of an old Odin Priority Queue:



Would be transformed into the new queue design:



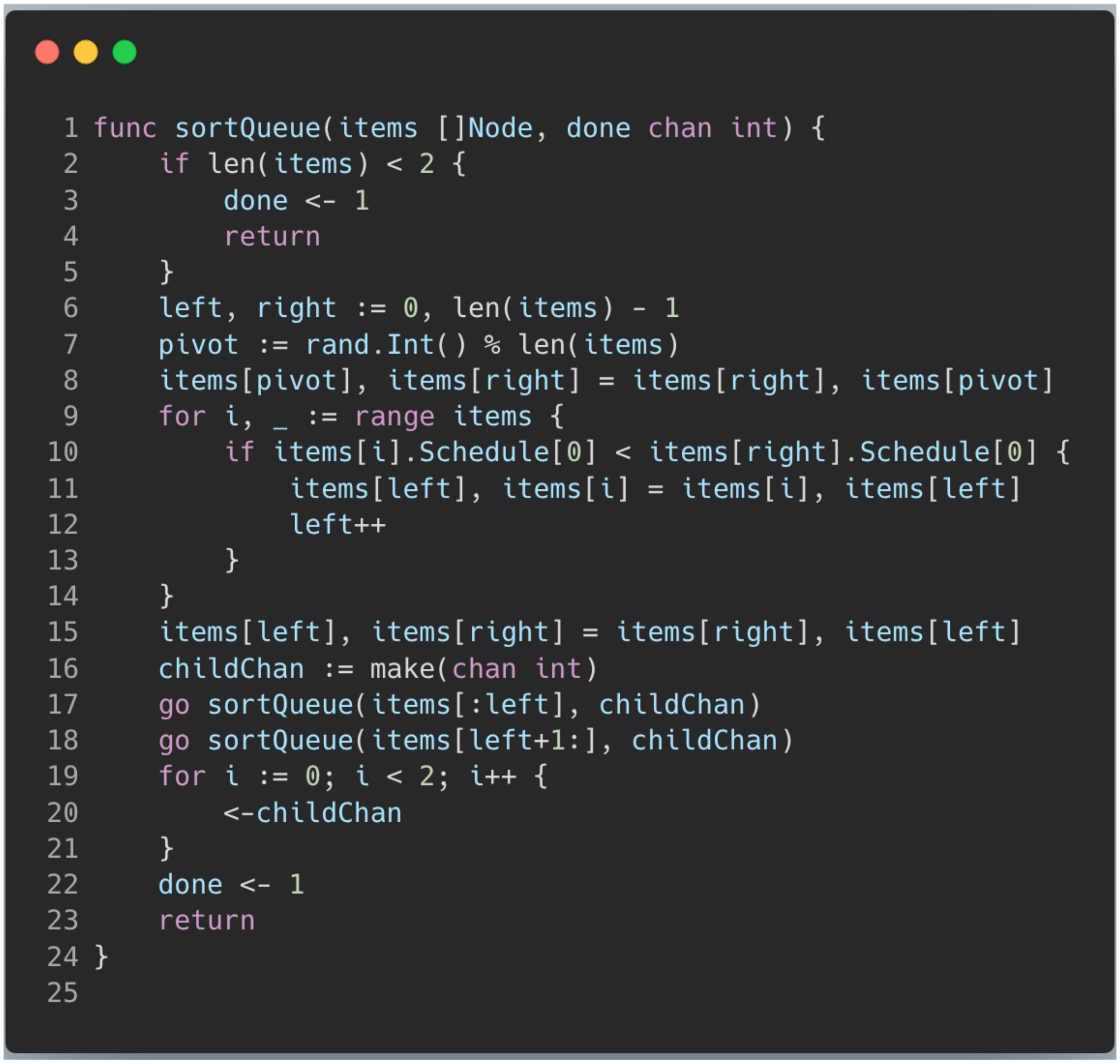
This now means that the problem of skipping over nodes which are ready to execute is an issue of the past, as execution is now run as a batch operation across the Jobs array of the node at the head of the queue.

4.2 Improving Concurrency

In this revised design, the process of sorting of Priority Queue was still quite slow. This needs to be a fast operation because the queue can only be grouped once it has been sorted. The grouping/mapping of jobs per node in the queue is a more computationally complex operation by default, but we also aimed to improve the speed of execution for this operation too.

While using a custom implementation of Quicksort, we knew we could leverage Go's built-in Goroutines to speed up our sorting algorithm. Goroutines are thin lightweight threads in the Go programming language which allow for the concurrent running of functions.

Our solution to the sorting problem can be seen in the code below:



```
 1 func sortQueue(items []Node, done chan int) {
 2     if len(items) < 2 {
 3         done <- 1
 4         return
 5     }
 6     left, right := 0, len(items) - 1
 7     pivot := rand.Int() % len(items)
 8     items[pivot], items[right] = items[right], items[pivot]
 9     for i, _ := range items {
10         if items[i].Schedule[0] < items[right].Schedule[0] {
11             items[left], items[i] = items[i], items[left]
12             left++
13         }
14     }
15     items[left], items[right] = items[right], items[left]
16     childChan := make(chan int)
17     go sortQueue(items[:left], childChan)
18     go sortQueue(items[left+1:], childChan)
19     for i := 0; i < 2; i++ {
20         <-childChan
21     }
22     done <- 1
23     return
24 }
25
```

The sorting algorithm above is a custom concurrent recursive implementation of Quicksort. Goroutines are acknowledged by the use of the go keyword, this essentially kicks off a new thread. As can be seen in lines 17 and 18, this means that both sides of the chosen pivot are executed simultaneously.

4.3 Making Odin a Distributed System

Scaling Odin out to be a distributed system was a significant challenge set out in the Odin Functional Spec. Distributing Odin would offer increased reliability to the system in case of a failure.

Transforming Odin into a distributed system required a significant amount of terraforming. We utilised the Hashicorp implementation of the Raft consensus protocol. The Raft protocol requires the election of a Leader node from the pool of candidates. A finite state machine was designed to allow all started nodes to be added as voters in this election and in turn cast their votes to reach consensus.

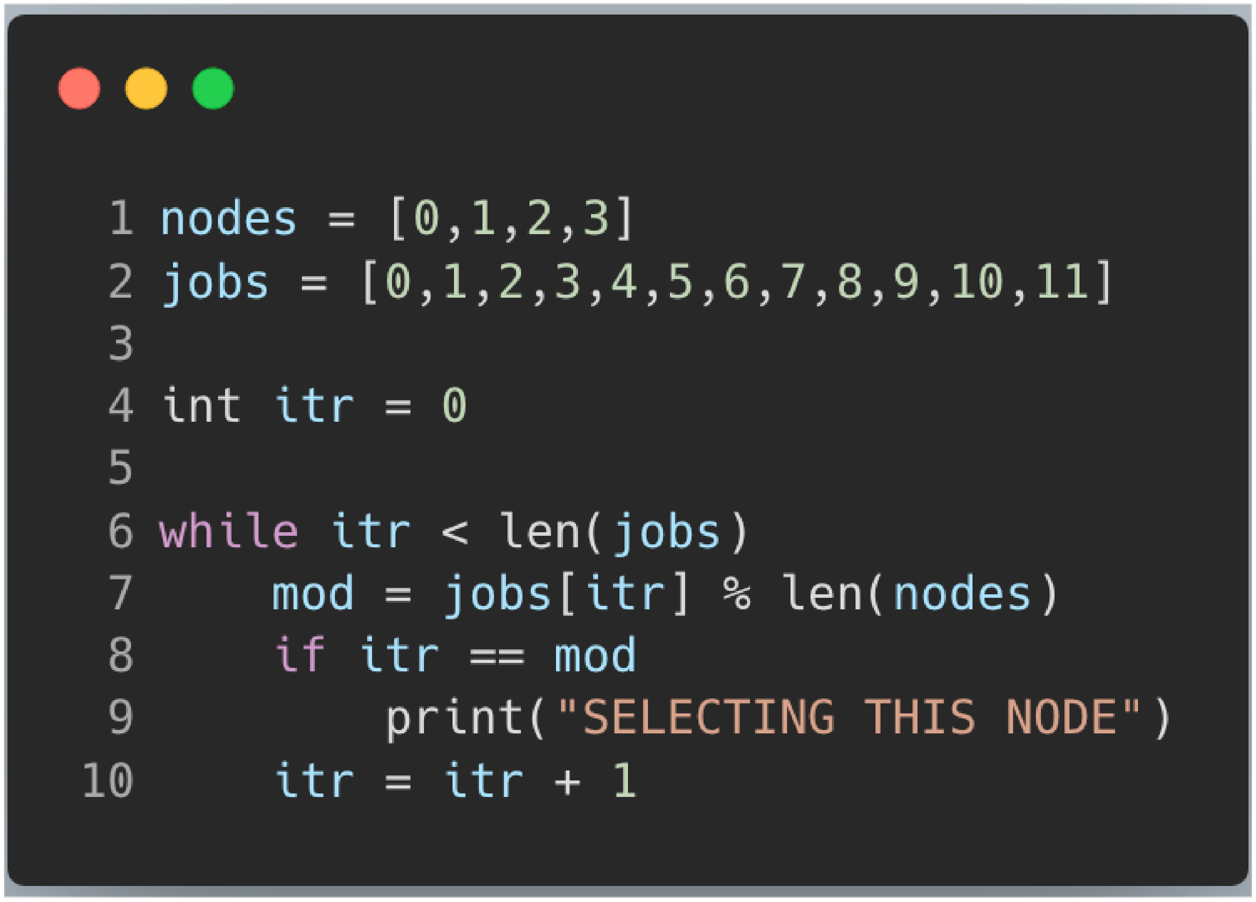
New API endpoints for `join` and `leave` were created so expanding the cluster was handled and nodes failing was handled. This did not prove to be a significant challenge but was necessary in managing the Raft Cluster also.

Another significant hurdle was deciding the way in which jobs were distributed to worker nodes. In designing a method of distributing work, we also needed to take into account how the work is distributed in the event of a node failing. In an example of 12 jobs and 4 nodes, ideally each node should execute 3 jobs each. If a node fails, then each node should take on some extra responsibility. These 3 remaining nodes will then share the 12 jobs by executing 4 jobs each.

Thus we reached a consensus of our own in how work was shared between nodes:

- Each node is assigned an integer, in the example of 4 nodes above this would be an integer between 0 and 3.
- Each job is then assigned an integer, so in the case of the twelve jobs an integer between 0 and 11.
- We then get the modulus of the current job integer (0-11) and the number of nodes in the cluster (currently 4). We call this `mod`.
- We then compare the node integer (0-3) to `mod` and if they are the same then that node is selected to execute the current job.

In pseudocode, the operation looks like this:



```
nodes = [0,1,2,3]
jobs = [0,1,2,3,4,5,6,7,8,9,10,11]

int itr = 0

while itr < len(jobs)
    mod = jobs[itr] % len(nodes)
    if itr == mod
        print("SELECTING THIS NODE")
    itr = itr + 1
```

5. Results

5.1 Problems Solved

In all, Odin solves the problem of periodic job scheduling in a way not currently available to the open source community. Odin is both an observable and a programmable orchestration system for the period scheduling and execution of user defined jobs, something which is to the best of the team's knowledge, not yet existent in the world of open source.

Odin provides automated configuration, coordination and management of user jobs with layers of abstraction sitting between the user and the job being executed, making it hard to interrupt the work done by the orchestration system. This solves the problems associated with schedulers like cron, which can fail at various hurdles due to user interaction.

As stated multiple times, Odin is deeply concerned with the conditions before and after the time of execution. Odin provides the observability of jobs through centralised logs and metric visualisation. Written in Python, Go and Node.js are the Odin Software Development Kits which all for this power, with the Odin Observability Dashboard centralising all aforementioned job information/metrics. We feel that the observability features are a significant achievement as it reduces any associated toil with debugging a job which has broken, helping infer the internal state of jobs and any given time.

To summarise, Odin is a first of its kind in the world of open source. It was built due to the absence of a true open source alternative to job orchestration that wasn't built on top of existing orchestration technologies like Docker Swarm, Kubernetes or Apache Mesos. That gap in the market is now fit by Odin. Odin also solves significant problems related to developer toil, making the debugging process significantly faster due to its one of a kind observability capabilities.

5.2 Testing

5.2.1 Testing Strategy

During development of the Odin system the team decided that an agile approach would work well due to our small size of two. This meant designing a sprint plan, with each sprint allocated to a feature or set of features. This contributed towards our selection of testing strategy - test-driven development (TDD).

Test-driven development is a strategy which relies on the repetition of a very short development cycle: requirements are turned into very test cases, then the code is improved so that the tests pass. The idea here is to write your test/test suite which fails before you write any new functional code. This means that for any new features, tests must be included and prioritised.

This strategy would work for the purposes of new features, but this was restricted to the realm of unit tests alone. As the project was time sensitive we could not write sprawling integration tests before developing every new feature. Integration tests written towards the end of development reflect this prioritisation in our testing, which has allowed us to speed up our development process.

We also implemented regression testing through the use of the Gitlab CI facility. Regression testing is generally considered to be the re-running functional and non-functional tests to ensure that previously developed and tested software still performs after a change. Continuous integration is integral to this process as we could leverage our Gitlab CI pipelines (set up for various branches on the project) to re-run our testing for each and every new change, ensuring each change was compatible with the last.

A portion of the testing is written in Go, which has fantastic built in tooling to check code coverage and apply benchmarking where necessary. All testing written in Go adopted a set of policies set out in a blog written by Hashicorp founder Mitchell Hashimoto. The blog post in question can be read here - https://about.sourcegraph.com/go/advanced-testing-in-go. The post encourages the use of what are known in Go as “Table Driven Tests”. This method of writing test cases:

- Yields low overhead in adding new test cases
- Makes testing exhaustive scenarios simple. It's easy to see visually if you've covered all cases.
- Makes reproducing reported issues simple

5.2.2 In Scope

During development of the system we acknowledged the following areas of the system to be in scope to testing:

- Odin-Engine
- Odin CLI
- Odin Software Development Kits
- Odin Observability Dashboard
- Integration and Regression Testing with Gitlab CI
- General Code Coverage

5.2.3 Out of Scope

During development of the system we acknowledged the following areas of the system to be out of scope to testing:

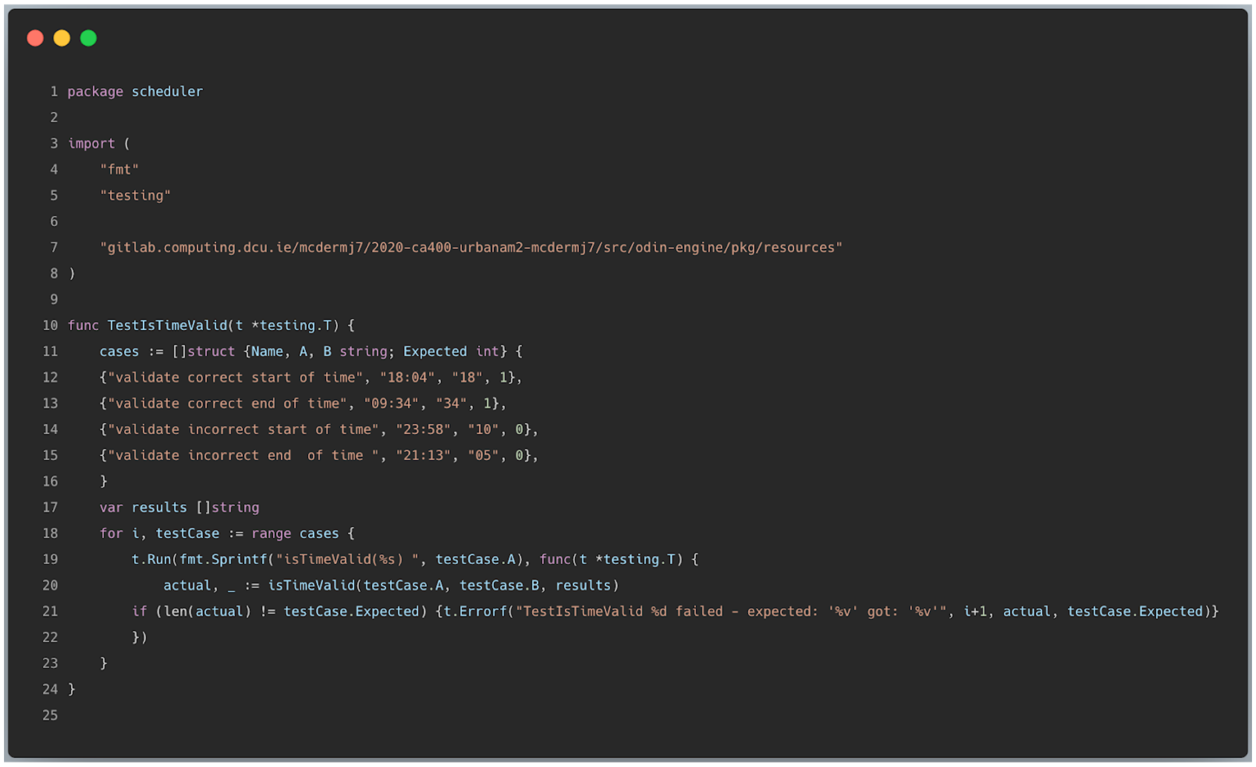
- MongoDB
- Third Party Dependencies

5.3 Types of Testing

5.3.1 Odin Engine - Unit Tests

All Odin Engine unit tests were written in Go and in accordance with the concept of Table Driven testing. Endpoints used for scheduling and managing jobs have been tested on a function by function basis.

These unit tests ensure that core engine functionality persists and were run as part of the continuous integration of code into the dev/master branch, which the team used as a pseudo-master branch before merging the final product into the master branch.



```
 1 package scheduler
 2
 3 import (
 4     "fmt"
 5     "testing"
 6
 7     "gitlab.computing.dcu.ie/mcdermj7/2020-ca400-urbanam2-mcdermj7/src/odin-engine/pkg/resources"
 8 )
 9
10 func TestIsTimeValid(t *testing.T) {
11     cases := []struct {Name, A, B string; Expected int} {
12     {"validate correct start of time", "18:04", "18", 1},
13     {"validate correct end of time", "09:34", "34", 1},
14     {"validate incorrect start of time", "23:58", "10", 0},
15     {"validate incorrect end  of time ", "21:13", "05", 0},
16     }
17     var results []string
18     for i, testCase := range cases {
19         t.Run(fmt.Sprintf("isTimeValid(%s) ", testCase.A), func(t *testing.T) {
20             actual, _ := isTimeValid(testCase.A, testCase.B, results)
21         if (len(actual) != testCase.Expected) {t.Errorf("TestIsTimeValid %d failed - expected: '%v' got: '%v'", i+1, actual, testCase.Expected)}
22         })
23     }
24 }
25
```

Above is a snippet from the tests run on the scheduler endpoint. The table of cases can be seen from lines 11 to 16, detailing expected inputs, output and results

These tests can be run using the `make test` command in the `src/odin-engine` directory of the project.

5.3.2 Odin CLI - Unit Tests

All Odin CLI unit tests were written in Go and in accordance with the concept of Table Driven testing. Unit tests were written to cover the primary invocation of each command along with and designed helper functions which facilitate this functionality.

These unit tests help to preserve the integrity of the tool which acts as the gateway to the Odin Engine. Running these tests requires the Odin Engine to be actively running as otherwise the team felt it would not demonstrate the integrity of the Odin CLI as a user facing tool.

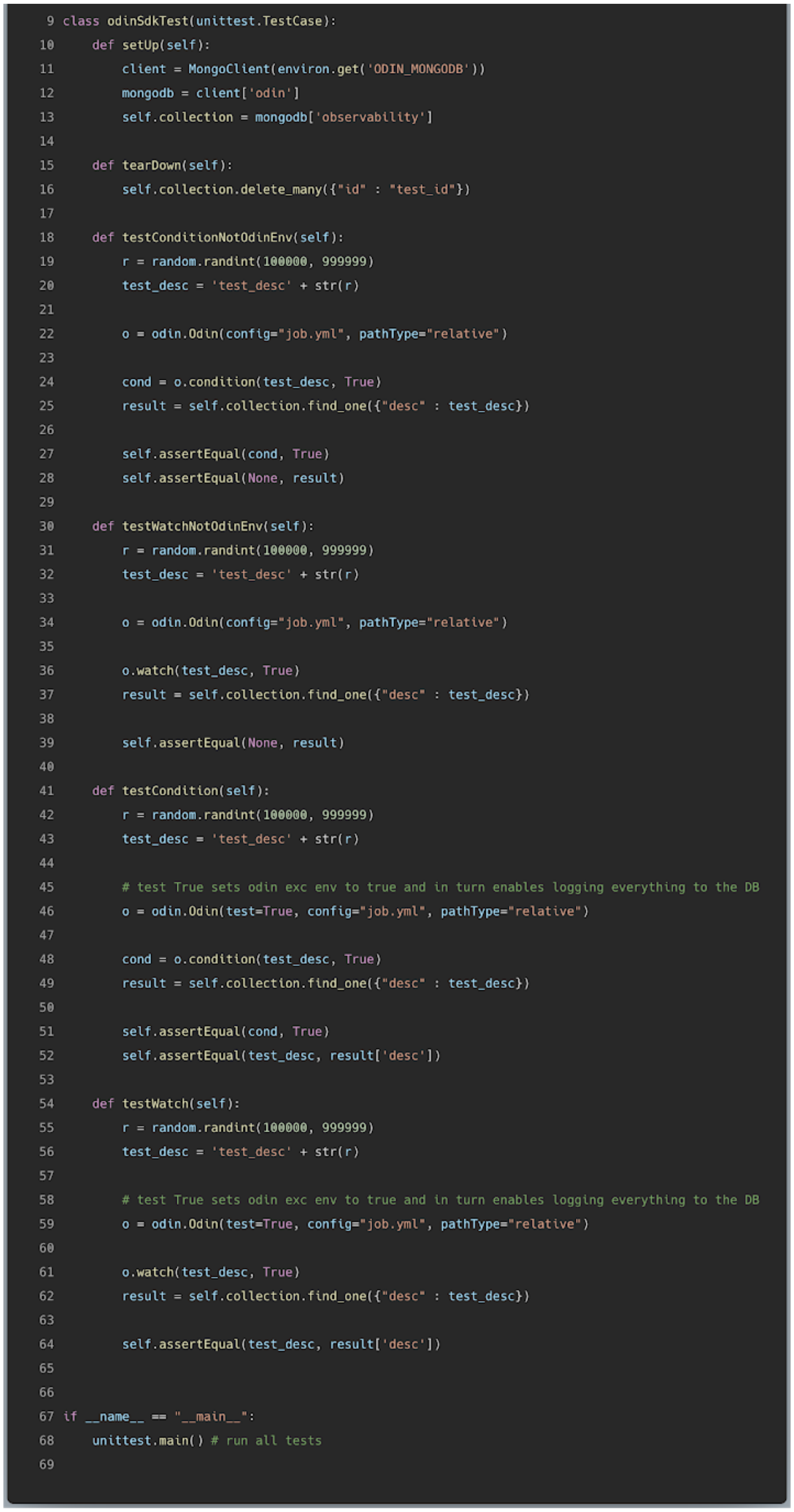
These tests can be run using the `make test` command in the `src/odin-cli` directory of the project.

5.3.3 Odin Software Development Kits - Unit Tests

The Odin Software Development Kits are written in three separate languages, Go, Node.js and Python. This requires testing to be language specific also. Unit tests were written in each language to ensure the logging functionality of each SDK is working in a controlled setting.

The use of environment variables and initialization variables for structures and objects allow us to mock the environment of the Odin Engine to ensure the integrity of the tests is as high as possible.

The below image represents the test cases carried out on the Python SDK, with specific focus on the watch, condition and result operations:



```python
class odinSdkTest(unittest.TestCase):
    def setUp(self):
        client = MongoClient(environ.get('ODIN_MONGODB'))
        mongodb = client['odin']
        self.collection = mongodb['observability']

    def tearDown(self):
        self.collection.delete_many({"id" : "test_id"})

    def testConditionNotOdinEnv(self):
        r = random.randint(100000, 999999)
        test_desc = 'test_desc' + str(r)

        o = odin.Odin(config="job.yml", pathType="relative")

        cond = o.condition(test_desc, True)
        result = self.collection.find_one({"desc" : test_desc})

        self.assertEqual(cond, True)
        self.assertEqual(None, result)

    def testWatchNotOdinEnv(self):
        r = random.randint(100000, 999999)
        test_desc = 'test_desc' + str(r)

        o = odin.Odin(config="job.yml", pathType="relative")

        o.watch(test_desc, True)
        result = self.collection.find_one({"desc" : test_desc})

        self.assertEqual(None, result)

    def testCondition(self):
        r = random.randint(100000, 999999)
        test_desc = 'test_desc' + str(r)

        # test True sets odin exc env to true and in turn enables logging everything to the DB
        o = odin.Odin(test=True, config="job.yml", pathType="relative")

        cond = o.condition(test_desc, True)
        result = self.collection.find_one({"desc" : test_desc})

        self.assertEqual(cond, True)
        self.assertEqual(test_desc, result['desc'])

    def testWatch(self):
        r = random.randint(100000, 999999)
        test_desc = 'test_desc' + str(r)

        # test True sets odin exc env to true and in turn enables logging everything to the DB
        o = odin.Odin(test=True, config="job.yml", pathType="relative")

        o.watch(test_desc, True)
        result = self.collection.find_one({"desc" : test_desc})

        self.assertEqual(test_desc, result['desc'])


if __name__ == "__main__":
    unittest.main() # run all tests

```

The Go tests can be run using the `go test` command in the `src/odin-libraries/go/odinlib` directory.

The Node.js tests can be run using the `npm test` command in the `src/odin-libraries/nodejs` directory.

The Python tests can be run using the `python3 odin_test.py` command in the `src/odin-libraries/python` directory.

5.3.4 Odin Observability Dashboard - Unit Tests

Odin Observability Dashboard back end server unit tests were written in javascript and used mocha while front end angular tests were written in Typescript using karma to test components.

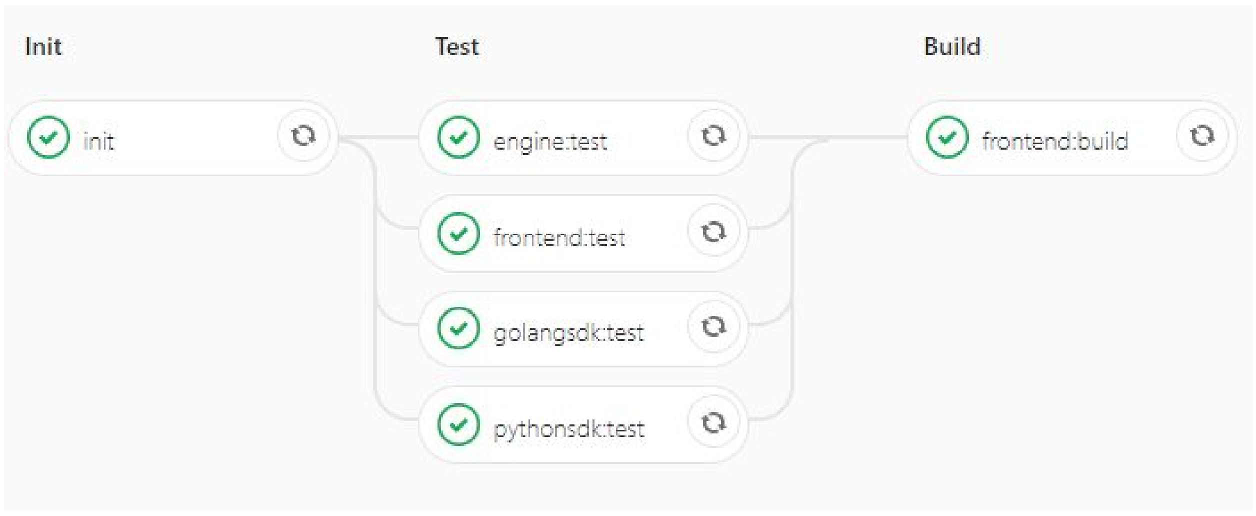
The unit tests ensure that the correct functionality is achieved and persists throughout development when paired with continuous integration.

Front end tests can be run using the `npm test` command in `src/webapp/client` directory.

Back end tests can be run using the `npm test ./tests` command in `src/webapp/server`

5.3.5 Integration and Regression Testing with Gitlab CI

As previously mentioned in our testing strategy and throughout this testing section, the team made good use of the Gitlab Continuous Integration pipelines to ensure new features introduced no breaking changes to existing code.



Our configured CI pipeline is shown above, it has 3 stages. Init stages sets up the nodeJS cache to be used for testing and building. After All test jobs are run including engine tests, dashboard frontend test, go SDK tests, python SDK tests. Observability dashboard front end build runs if the previous tests were successful.

This regression testing was run on all aspects the team considered to be in scope, including:

- Odin-Engine
- Odin Software Development Kits
- Odin Observability Dashboards

This ensured that changes to all of the above components were integrated as production ready when they passed out pre-written tests. Each time a pipeline was run, it did so with the following set of pre-configured global variables used throughout the pipeline:

```
variables:
  CLI_VERSION: 6.1.5
  GO_VERSION: 1.13.7
  DIST_DIR: dist/
  REPO: gitlab.computing.dcu.ie
  GROUP: mcdermj7
  PROJECT: 2020-ca400-urbanam2-mcdermj7
```

The various stages of testing (init, test, build) can be viewed in the `.gitlab-ci.yml` file in the root directory of the project repository.

The pipeline status of the Odin project may be viewed here: https://gitlab.computing.dcu.ie/mcdermj7/2020-ca400-urbanam2-mcdermj7/badges/dev/master/pipeline.svg

5.3.6 General Code Coverage

Code coverage is generally considered to be a good metric of how rigorously tested a project is. Code coverage is the degree to which the source code of a program is executed when a particular test suite runs. This is worked out as a percentage of lines traversed during the execution of test suites.

The code coverage of the Odin project may be viewed here: https://gitlab.computing.dcu.ie/mcdermj7/2020-ca400-urbanam2-mcdermj7/badges/dev/master/coverage.svg

6. Future Work

6.1 The Odin Engine

Moving forward, the team believes that the Odin Engine and it's components could be made more modular in their architecture. Endpoints like the jobs and execute endpoints act as central hubs of logic nested within the API. While this does work in Odin's favour with respect to distribution, there could be better alternatives in regards to infrastructure.

Future work with the Odin Engine would involve researching microservice models and seeing what general benefits such models yield over distributed systems. From our point of view, a microservice model would make aspects of the Engine more composable.

6.2 The Odin CLI

Moving forward with the Odin CLI, the team recommends investing more time into the existing commands. While new functionality would be interesting, we feel more work can be invested in improving the Odin CLI capabilities.

Taking the "log" command as an example, future work can come in the form of adding flags to make the log command more powerful. Flags which allow for filtering by date and time would most certainly be a significant contribution, much more useful than any more new commands. The same applies to each existing command, it's likely that further maximisation of the commands potential yields more benefit than simply adding new commands

6.3 The Odin Software Development Kits

Recommending future work for the Odin Software Development Kits is simple - build more language specific libraries.

Currently, Odin supports jobs run in Python, Go and Node.js - languages such as Rust, Java and C++ could prove to be popular additions to the runtimes supported by Odin. These changes could potentially expand the system to entice more developers to use it.

6.4 The Odin Observability Dashboard

Recommending future work for the Odin Observability dashboard includes working on more integration with Odin Engine so as to allow a more user friendly experience and enhanced engine controls via the web facing user interface.

Visualization is a powerful tool, future work in regards to the user interface may also encompass making the interface a multi Linux user login based system also.