Group C - DevOpps
DevOps BSDSESM1KU

Alexander Hjelmgaard
hjel@itu.dk

Oskar Breindahl
osbr@itu.dk

Christoffer Gram
chkg@itu.dk

Jakob Hjalgrim
jlhj@itu.dk

Anton Folkmann
anpf@itu.dk

May 31, 2022

Contents

1 System Perspective **1**

1.1 Requirements . . . 1

1.2 Design and Architecture . . . 1

1.3 Dependencies . . . 2

1.4 Current State of the System . . . 4

1.5 Licensing . . . 4

2 Process Perspective **5**

2.1 The Team's Setup . . . 5

2.2 CI/CD Chains . . . 5

2.3 Repository organization . . . 6

2.3.1 Repositories . . . 6

2.3.2 Branching . . . 6

2.3.3 Development process/tools . . . 6

2.4 Monitoring . . . 7

2.5 Logging . . . 7

2.6 Security . . . 7

2.7 Scalability . . . 8

3 Lessons Learned Perspective **9**

3.1 Evolution and Refactoring . . . 9

3.1.1 Testing . . . 9

3.2 Operations . . . 9

3.2.1 Weekly Operations . . . 9

3.2.2 Limiting Work in Progress (WIP) and Batch Size with the GitHub Ecosystem . . . 9

3.2.3 The importance of update strategies . . . 9

3.3 Maintenance . . . 10

3.3.1 Live Software's impact on Software Maintenance . . . 10

3.3.2 The Immediate Benefit of Logging . . . 10

3.4 Server storage & Database management . . . 10

A *Minitwit's* Go Package Dependencies **13**

B *Minitwit's* Production Environment Dependencies **19**

C An Snapshot of the Project Board **23**

1 System Perspective

1.1 Requirements

The functional requirements are:

1. The *Minitwit* system shall have the same functionality as the original *Minitwit* system.
2. The *Minitwit* system shall expose an API that meets the *Simulator's* specification.

The non-functional requirements are:

1. The system shall be written in Go using the Gorilla framework and JavaScript using the React library.
2. The system shall be hosted on DigitalOcean Droplet instances.
3. The system shall be accessible by end-users independent of operating system.
4. Users with experience from other chat services such as Twitter should be able to post a tweet/message.

1.2 Design and Architecture

The requirements of the *Minitwit* system are well-defined. Hence, the design of the system is limited to the selection of the three-tier architecture. Figure 1 below depicts this architecture applied to the *Minitwit* system.

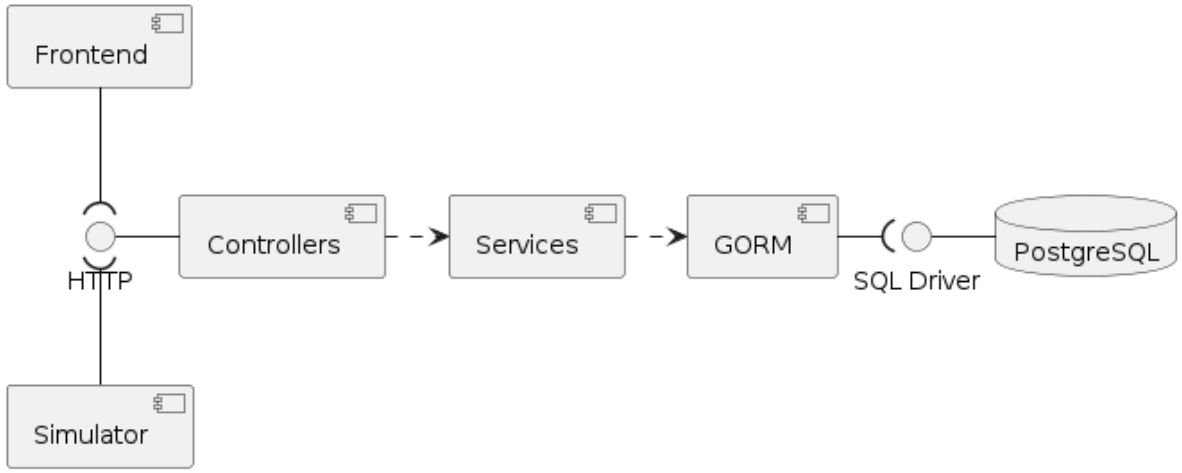

Figure 1: A high-level overview of *Minitwit's* three-tier architecture

Each layer is discretely bounded by communication interfaces. The first layer consists of two clients to the *Minitwit* system. The frontend allows end-users to interact with the system, and the simulator replicates live users interacting with the system. The second layer provides the core functionality of the *Minitwit* system. It exposes this functionality through two HTTP APIs, for the frontend and simulator respectively. The third layer is a PostgreSQL database that provides the system with persistent data.

To implement the *Services* component there is an important interaction with the *GORM* component. *GORM* is an object-relational mapper for Golang. The *Services* component uses it as a bridge pattern to add a layer of abstraction such that several database vendors can provide persistence functionality.

Figure 2 below reveals how the use of different SQL drivers allows *Minitwit* to support different database vendors.

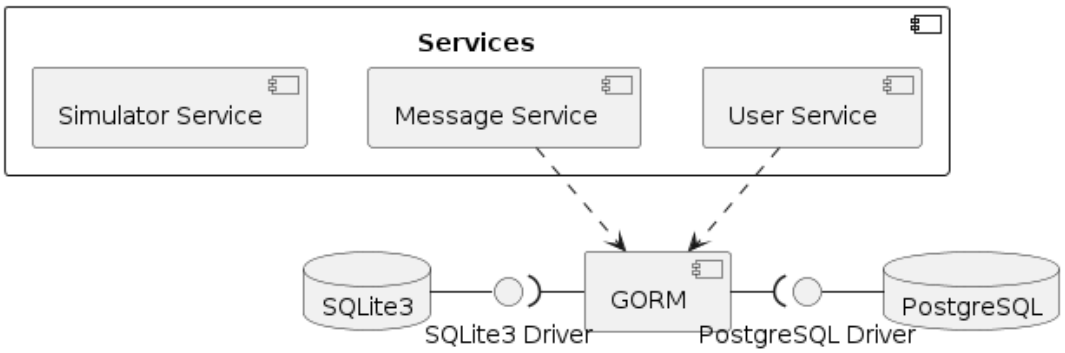

Figure 2: An illustration of *Minitwit's* bridge pattern to support different database vendors

1.3 Dependencies

To boost developer productivity the *Minitwit* system uses Go's own dependency management system to import third-party packages. A noteworthy package we depend on is the *Gorilla* package. It provides functionality to implement the HTTP APIs required by the functional requirements, such as a multiplexer and session management. Figure 3 overleaf is a visualization of the Go package dependencies of *Minitwit* with a depth of one. We have also generated a complete list and complete visualization (see Appendix A).

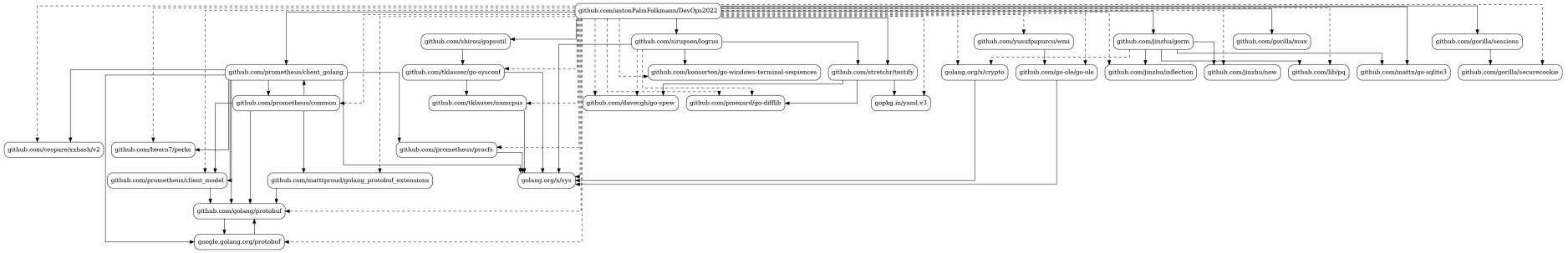

Figure 3: An graph of Minitwit's immediate dependencies. A full resolution can be found here here.

To package the application we depend on Docker. To provision environments we use DigitalOcean's CLI tool *doctl*. By using Docker we can encode dependencies in the form of layers for the image. Notably, the base layer of the *Minitwit* system is the Golang 1.17 image. We take advantage of *doctl*'s ability to programmatically provide environments with Docker preinstalled. In this way we can be sure that any environment we provision can run the *Minitwit* application, since it is packaged with Docker. A complete list of our production environments dependencies can be found in Appendix B.

1.4 Current State of the System

The system has evolved to be more scalable, maintainable, and reliable. The system can scale with multiple users sending simultaneous requests by taking advantage of Gorilla's multi-threaded multiplexer. Scalability and reliability is delivered by replicating the *Minitwit* system across multiple physical nodes using Docker swarm. Additionally, we have introduced continuous integration (CI) pipelines to ensure the reliability of the system. Finally, the system has been made more maintainable. The system is observable through features such as logging and monitoring, and updating the system has been eased through our continuous delivery (CD) pipeline.

The maintenance work is still not complete, and some top priority improvements are:

1. Implement the ability to flag tweets.
2. Update Go dependency to the latest version (to Golang 1.18).
3. Add more indexed fields log messages to add more granularity when filtering and searching for log messages.

1.5 Licensing

During the development of the application the project was licensed under the Massachusetts Institute of Technology (MIT) open-source license. The motivation for using the MIT license was its lack of restrictions regarding distribution, along with its compatibility with many other licenses. Most of the application's direct dependencies' licenses are permissive licenses such as BSD, Apache, ISC and other MIT licenses. However, during the report writing phase of the project, it has come to the team's attention that some dependencies are licensed under the Mozilla Public License (MPL). Due to the MPL being classified as a weakly protective license, licensing the project under the MIT license is not restrictive enough. As a result, the project should be re-licensed under the MPL to comply.

2 Process Perspective

2.1 The Team's Setup

Interaction between team members is a hybrid of physical meetings onsite at ITU and online meetings via Discord.

During the project's preliminary phase, the team defined guidelines for a distributed workflow to be followed. The guidelines address how the team utilizes Trunk Based Development [3], defines when a contribution is considered done, ascribes responsibility for integrating and reviewing these contributions, and how to document knowledge acquisition.

The workflow comprises of three phases; Planning, development, and release. In the first two phases we prefer synchronous communication to simplify coordination. The team discusses, defines, and distributes tasks for completing the week's working during the planning phase. Thereafter, the tasks are managed on a GitHub project board (see Appendix C). In the development phase tasks are delegated to one to two members. When possible pair programming is used to be more productive [1]. Hence, the team has a clear overview of everyone's tasks and their progress during the development and release. This reduces the need for communication as a method of coordination since the project board is a single source of work.

2.2 CI/CD Chains

Figure 4 below shows the series of quality gates that are part of the CI pipeline.

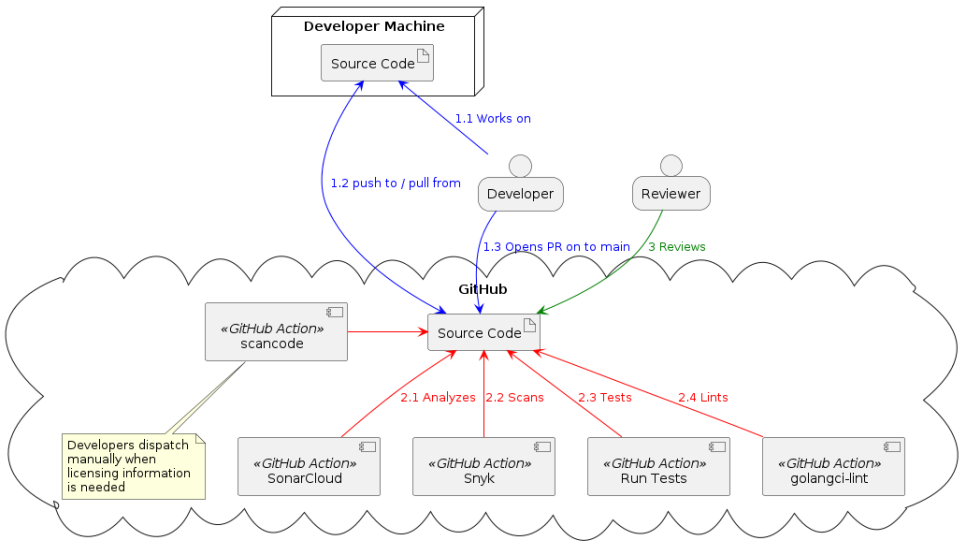

Figure 4: A Diagram of *Minitwit's* CI/CD pipeline

We employ quality gates to ensure high levels of code quality and security. First, we use Git and GitHub for version control in a distributed work setting. Once a developer is satisfied with their work, they open a pull request to merge the changes into main. There are four quality gates that a PR must meet before it can be merged. The pull request must then be reviewed and approved by another developer. This bakes code reviews into our way of working. Additionally, golangci-lint and SoncarCloud provide different perspectives on the quality of the code. Furthermore, Snyk scans the repository for security vulnerabilities providing increased confidence that the code is safe for production. Finally, the PR must

also pass the test suite. Although not automated, we have made it effortless to gain licensing information through the scancode GitHub action.

Figure 5 below shows the build and deployment steps that are part of our CI/CD pipeline.

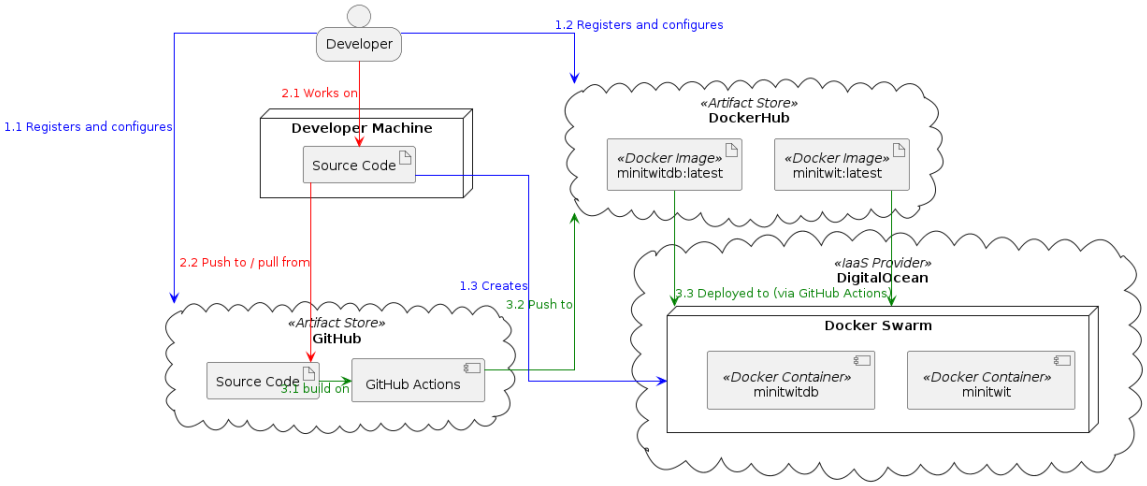

Figure 5: A Diagram of *Minitwit's* CI/CD pipeline

When a new feature is on the main branch, it will be included in the next weekly release. When we make a release on GitHub, the main branch is built using GitHub actions and packaged into docker images. Dockerhub acts as an artifact store for these images. Then GitHub actions will act on the Swarm Leader to pull the new images into the Docker Swarm using a rolling update strategy.

2.3 Repository organization

2.3.1 Repositories

The team chose to work with a single mono-repository hosted on GitHub. Later in the project, a separate repository was introduced for developing the React front-end to ensure low coupling to the back-end. The content of this repository were eventually inserted into the original repository when it was deemed sufficiently sophisticated. The introduction of an additional repository temporarily derailed the mono-repository approach. The team contemplated changing the repository approach by introducing an organization, but decided to continue with a mono-repository for the remainder of the project.

2.3.2 Branching

The team chose to disallow pushes to the main branch, meaning all contributions needed a separate branch and a subsequent pull request. As mentioned in section 2.2, pull requests were set up to require at least one approved review along with passing all tests to be merged into main.

2.3.3 Development process/tools

Initially, GitHub issues were used to track and assign tasks, but it became apparent that a more sophisticated approach was needed. The team then chose to create a GitHub project board with 5 swim lanes (*Backlog*, *Ready to Do*, *In Progress*, *Ready to Review*, *Done*). Additionally, we chose to limit the *In Progress* column to 10 issues, to make sure our pace remained consistent and issues didn't end up hanging for too long.

2.4 Monitoring

The application is actively monitored utilizing Prometheus with the metrics visualized in Grafana. We utilized Whitebox monitoring by recording and logging events and response times happening inside the system when simulating user activity with HTTP requests and displaying them with monitoring software. Furthermore, Blackbox monitoring of server CPU usage is performed to determine if the server CPU is wasting resources or has enough resources to be able to handle more jobs. When applying the three-level maturity model [4], the monitoring stage of the application can be identified within the "reactive" stage. The aforementioned components monitored provide basic measurements of the performance of the application along with an insight of the user experience in terms of accessibility and responsiveness. The resulting metrics are transformed such that they can be displayed in graphs, gauges and counters using Grafana.

2.5 Logging

Our system uses the Go package *logrus* for most of its logging. Logging mainly takes place in the service layer of the application, with extensive logging taking place in the simulator and frontend endpoints. As a rule, whenever there is a call through the tiers, i.e. from frontend to backend to database, the systems services handles logging, printing statements or warnings based on the action's results.

To index and present logs in a developer-friendly manner, we use a stack consisting of filebeat, elastic-search, and kibana. Filebeat parses the logs into JSON, and transmits the JSON to elastic-search. Elastic-search maintains the logs in a structured manner such that they can be indexed. Finally, Kibana acts as a front end to Elastic-search enabling developers to filter and read log messages.

2.6 Security

Our assets include the Docker containers, client information, developer machines and the tools we use. Threat sources to our assets include system failure (if our tools went down), errors made by the developers or malicious attacks on the software.

Concerning the above analysis, we have constructed 5 risk scenarios for our product posed below:

- SQL injection: A user with malicious intents performs SQL injection using the login page/add message page to steal sensitive information and drop tables. (Scenario 1)
- Accidental human interference:
  - A developer writes faulty code, which makes it to production and exposes sensitive data. (Scenario 2)
  - A developer writes code that wrongly handles database queries, resulting in loss of data. (Scenario 3)
- System failure:
  - DigitalOcean: The DigitalOcean service crashes, causing all web applications to fail. (Scenario 4)
  - Snyk: While attempting to merge code with security vulnerabilities into the main branch, Snyk crashes and the vulnerabilities make it to production. (Scenario 5)

Then we have analyzed the 5 scenarios according to impact and likelihood as shown in table 1.

Impact \likelihood	Low	Medium	High
Low		Scenario 2 and 3	
Medium			
High	Scenario 1, 4, 5		

Table 1: Risk matrix

Thus, there are no threats that are in the critical zone (high/high). As such, the system is reasonably safe as the likelihood of risk is very low for high impact threats or are of low impact.

2.7 Scalability

Terraform creates a distributed cluster of droplets on DigitalOcean and includes them in a Docker Swarm. We then use Docker Swarm to manage replication of the application across the distributed cluster. We do this to delegate workload among multiple nodes and increase resilience with more fault tolerance. The Docker Swarm stack runs 10 replica nodes of the *Minitwit* webserver. However, since the database is stateful, we only deploy one replica.

3 Lessons Learned Perspective

3.1 Evolution and Refactoring

3.1.1 Testing

Testing during the refactorization of an existing system is highly valuable. When refactoring you may inadvertently change the functionality of a system. If not discovered early this could lead to bugs and hours of work trying to fix them. For example, during the refactorization of the simulator-API from Python to Golang, we introduced a tweet bug. The bug could have been avoided if we had tested more thoroughly before merging the refactored simulator-API. Each sub-component was unit tested, as such we naively thought everything was working as intended. However, unit tests do not necessarily tell if the system as a whole is working as intended. In reality, the tweet endpoint did not give the expected responses. What we should have done differently, was run the simulator against the API or have done more manual tests locally, checking for inconsistencies. The process as a whole mimics well what could happen in real-world situations and has been a very valuable lesson learned. Going forward, we know that integration testing is not only important but necessary to deliver safe services.

3.2 Operations

3.2.1 Weekly Operations

Developers follow a weekly schedule, with an average of 1 workday/week/person being allocated to new tasks, management, and upkeep of the system. This effort includes making sure the simulator is able to use our system, making sure our logging implementation is producing the desired result and monitoring processing/storage load. The operations aspect of DevOps takes a backseat in our workflow as the team consists of inexperienced developers primarily focused on producing code but not operating it. Most of our energy is spent on maintenance and refactoring of the system. However, the group anticipates that the operational tools and practices taught in the course will be invaluable to our eventual work in the industry, where software is rarely greenfield and almost always live.

3.2.2 Limiting Work in Progress (WIP) and Batch Size with the GitHub Ecosystem

We use GitHub issues and GitHub project boards to successfully track our work. A key aspect of Lean, and by extension DevOps, is reducing batch sizes and the amount of WIP. We adopt these philosophises as they are presented in the The DevOps Handbook [2] wholesale. Tasks are decomposed to reduce batch size and a maximum of 10 issues are allowed in the "In Progress" column (see Appendix C). To clearly communicate what work is part of resolving an issue, we link pull requests to issues. The consensus is that this lightweight, transparent approach to distribute work enables flexibility. We believe that the efficacy of this will only increase as work schedules increase from sporadic to daily.

3.2.3 The importance of update strategies

We have learned that update strategies are an important factor in reliable services in real-world applications. During the first half of the course we had neither update strategy nor a horizontally scaled service. In order to update our service, we had to take down the service, deploy the changes and reboot the server. By implementing Docker Swarm we have started scaling the service horizontally. It allows us to deploy a rolling update strategy to our service, deploying updates without shutting it down. Before adopting an update strategy, we acknowledged that shutting down the service to update is far from optimal, but

after implementing the Rolling update strategy, we realize how beneficial it is to deploy and how much easier maintaining the program is. We note from this that employing an update strategy, even in the early stages of a service's development, is important. It keeps the service running with minimal downtime and reduces risk of outages. It helps to apply Lean and DevOps practices by automating processes and reducing batch size for updates. Furthermore, it simplifies and codifies the team's approach to updates.

3.3 Maintenance

3.3.1 Live Software's impact on Software Maintenance

Developing and extending a running system is a unique challenge. Minor changes in the code can lead to client requests failing. As mentioned in section 3.2.1, a bug was introduced halfway through the project. Issues like this highlight the difference between working in a static environment versus a live one, and how fragile the system can be. Additionally, the task of incorporating new technologies, like Terraform, became bloated. Tasks like these were further complicated by the lack of proper integration testing in our pipeline. A few weeks from the simulator's start, we realized that our database was not compatible with our CI/CD pipeline. This meant shutting down the server and containerizing a new database based on the initial database given at the start of the project. This issue is most likely not one to occur in a static (non-live) environment, but was is an experience that exposed the need of having a database management strategy early. These issues mimic real-world challenges of live software development, and give us valuable lessons we can learn from.

3.3.2 The Immediate Benefit of Logging

Logging turned out to be a bigger challenge than anticipated. With little to no experience before the project, an advanced tool like Kibana in junction with Elasticsearch was overwhelming but also insightful. We faced a couple of hurdles before successfully implementing logging. We found it difficult to locate the logs in a docker container as well as produce usable logs from our code. Once implemented, it was obvious how powerful logging is. Maintenance and error handling became easier. For example, the frontend was easier to implement, even with a live backend. The logs helped sort out eventual issues with REST-requests or database queries. From this, we noted logging should be implemented as early as possible, be it in a Greenfield project or an overtaken one like this. It brings insight into the system's functionality and is especially valuable when refactoring the code-base and writing new functionality. Though logging also poses the problem of producing large amounts of data. To ensure our container didn't flood with unnecessary data, we implemented log rotation and pruning. We learned that logging is a powerful tool that eases many processes, but it also requires attention to the level of detail to be useful.

3.4 Server storage & Database management

A non-trivial challenge of a live system is managing its persistent state such that data is not lost. Initially, volumes were not properly mounted to our database container. As a result, the next time the system was restarted data about messages and users was lost. From this experience, it is clear that it is important to gain a deep understanding of a technology before employing it in production. Moreover, we host all three layers - frontend, backend, and persistency - on the same infrastructure. Thus, to tear down production infrastructure we must tear down the system in its entirety. This means that any data not on a back-up is lost. The issue stems from not separating concerns properly, meaning that there is no fine grain control of the system. We believe there are two possible solutions to this:

- Accept that we do not have this granularity and employ a back-up strategy to mitigate the damages of data loss.
- Design the infrastructure such that this granularity exists

Ultimately, this is our first experience with a "live" state where the knock-on effects of data loss is observable, and it opens our eyes to the importance of mitigating data loss from the beginning.

References

[1] L. Williams. "Strengthening the case for pair programming". **in***IEEE Software*: 17.4 (2000), **pages** 19–25. DOI: 10.1109/52.854064.

[2] J. Humble. "Part I - The Three Ways". **in***The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations*: Portland, OR, USA: IT Revolution Press, 2016.

[3] P. Hammant. *Introduction*. Trunk Based Development. last accessed 27/05/2022. 2020 [Online]. URL: https://trunkbaseddevelopment.com/#one-line-summary.

[4] J. Turnbull. *Reactive*. A Monitoring Maturity Model. last accessed 27/5/2022. 2015 [Online]. URL: https://www.kartar.net/2015/01/a-monitoring-maturity-model/.

A *Minitwit's* Go Package Dependencies

A complete dependency graph can be found here. It is a visualization of the list below.

Below is the result of executing `go list -deps all` for the *Websever's* Go module. This produces a list of all the dependencies of the program in a depth-first, post-order ordering. This means that a given Go package is only listed after all of its dependencies have been listed. The exact versions are encoded in the *go.mod*[1] and *go.sum*[2] files of the *Webserver* Go module.

- unsafe
- internal/unsafeheader
- internal/abi
- internal/cpu
- internal/bytealg
- internal/goexperiment
- runtime/internal/atomic
- runtime/internal/sys
- runtime/internal/math
- runtime
- internal/reflectlite
- errors
- internal/race
- sync/atomic
- sync
- io
- unicode
- unicode/utf8
- bytes
- encoding
- math/bits
- math
- internal/itoa
- strconv
- reflect

[1] *go.mod* can be found at: https://github.com/antonPalmFolkmann/DevOps2022/blob/main/src/webserver/go.mod
[2] *go.sum* can be found at: https://github.com/antonPalmFolkmann/DevOps2022/blob/main/src/webserver/go.sum

- encoding/binary
- encoding/base64
- sort
- internal/fmtsort
- internal/oserror
- syscall
- internal/syscall/unix
- time
- internal/poll
- internal/syscall/execenv
- internal/testlog
- path
- io/fs
- os
- fmt
- strings
- unicode/utf16
- encoding/json
- hash
- crypto
- crypto/md5
- context
- crypto/sha1
- database/sql/driver
- database/sql
- regexp/syntax
- regexp
- github.com/jinzhu/inflection
- go/token
- path/filepath
- go/scanner
- go/ast

- log
- github.com/jinzhu/gorm
- bufio
- crypto/sha256
- container/list
- crypto/internal/subtle
- crypto/subtle
- crypto/cipher
- crypto/aes
- crypto/des
- crypto/elliptic/internal/fiat
- math/rand
- math/big
- crypto/elliptic
- crypto/internal/randutil
- crypto/sha512
- encoding/asn1
- vendor/golang.org/x/crypto/cryptobyte/asn1
- vendor/golang.org/x/crypto/cryptobyte
- crypto/ecdsa
- crypto/ed25519/internal/edwards25519/field
- crypto/ed25519/internal/edwards25519
- crypto/rand
- crypto/ed25519
- crypto/hmac
- crypto/rc4
- crypto/rsa
- crypto/dsa
- encoding/hex
- crypto/x509/pkix
- encoding/pem
- vendor/golang.org/x/net/dns/dnsmessage

- internal/nettrace
- internal/singleflight
- runtime/cgo
- net
- net/url
- crypto/x509
- vendor/golang.org/x/crypto/internal/subtle
- vendor/golang.org/x/crypto/chacha20
- vendor/golang.org/x/crypto/poly1305
- io/ioutil
- vendor/golang.org/x/sys/cpu
- vendor/golang.org/x/crypto/chacha20poly1305
- vendor/golang.org/x/crypto/curve25519
- vendor/golang.org/x/crypto/hkdf
- crypto/tls
- github.com/lib/pq/oid
- github.com/lib/pq/scram
- os/user
- github.com/lib/pq
- github.com/lib/pq/hstore
- github.com/jinzhu/gorm/dialects/postgres
- github.com/antonPalmFolkmann/DevOps2022/storage
- golang.org/x/sys/internal/unsafeheader
- golang.org/x/sys/unix
- github.com/sirupsen/logrus
- compress/flate
- hash/crc32
- compress/gzip
- vendor/golang.org/x/text/transform
- vendor/golang.org/x/text/unicode/bidi
- vendor/golang.org/x/text/secure/bidirule
- vendor/golang.org/x/text/unicode/norm

- vendor/golang.org/x/net/idna
- net/textproto
- vendor/golang.org/x/net/http/httpguts
- vendor/golang.org/x/net/http/httpproxy
- vendor/golang.org/x/net/http2/hpack
- mime
- mime/quotedprintable
- mime/multipart
- net/http/httptrace
- net/http/internal
- net/http/internal/ascii
- net/http
- github.com/antonPalmFolkmann/DevOps2022/services
- github.com/gorilla/mux
- github.com/antonPalmFolkmann/DevOps2022/utils
- encoding/base32
- encoding/gob
- github.com/gorilla/securecookie
- github.com/gorilla/sessions
- github.com/antonPalmFolkmann/DevOps2022/controllers
- expvar
- github.com/beorn7/perks/quantile
- github.com/cespare/xxhash/v2
- hash/fnv
- google.golang.org/protobuf/internal/detrand
- google.golang.org/protobuf/internal/errors
- google.golang.org/protobuf/encoding/protowire
- google.golang.org/protobuf/internal/pragma
- google.golang.org/protobuf/reflect/protoreflect
- google.golang.org/protobuf/internal/encoding/messageset
- google.golang.org/protobuf/internal/flags
- google.golang.org/protobuf/internal/strs

- google.golang.org/protobuf/internal/encoding/text
- google.golang.org/protobuf/internal/genid
- google.golang.org/protobuf/internal/order
- google.golang.org/protobuf/internal/set
- google.golang.org/protobuf/reflect/protoregistry
- google.golang.org/protobuf/runtime/protoiface
- google.golang.org/protobuf/proto
- google.golang.org/protobuf/encoding/prototext
- google.golang.org/protobuf/internal/encoding/defval
- google.golang.org/protobuf/internal/descfmt
- google.golang.org/protobuf/internal/descopts
- google.golang.org/protobuf/internal/filedesc
- google.golang.org/protobuf/internal/encoding/tag
- google.golang.org/protobuf/internal/impl
- google.golang.org/protobuf/internal/filetype
- google.golang.org/protobuf/internal/version
- google.golang.org/protobuf/runtime/protoimpl
- google.golang.org/protobuf/types/descriptorpb
- google.golang.org/protobuf/reflect/protodesc
- github.com/golang/protobuf/proto
- google.golang.org/protobuf/types/known/timestamppb
- github.com/golang/protobuf/ptypes/timestamp
- github.com/prometheus/client_model/go
- github.com/prometheus/common/model
- runtime/metrics
- github.com/prometheus/client_golang/prometheus/internal
- google.golang.org/protobuf/types/known/anypb
- github.com/golang/protobuf/ptypes/any
- google.golang.org/protobuf/types/known/durationpb
- github.com/golang/protobuf/ptypes/duration
- github.com/golang/protobuf/ptypes
- github.com/matttproud/golang_protobuf_extensions/pbutil

- github.com/prometheus/common/internal/bitbucket.org/ww/goautoneg
- github.com/prometheus/common/expfmt
- github.com/prometheus/procfs/internal/fs
- github.com/prometheus/procfs/internal/util
- github.com/prometheus/procfs
- runtime/debug
- github.com/prometheus/client_golang/prometheus
- github.com/prometheus/client_golang/prometheus/promhttp
- os/exec
- github.com/shirou/gopsutil/internal/common
- github.com/tklauser/numcpus
- github.com/tklauser/go-sysconf
- github.com/shirou/gopsutil/cpu
- github.com/antonPalmFolkmann/DevOps2022/monitoring
- github.com/antonPalmFolkmann/DevOps2022

B *Minitwit's* Production Environment Dependencies

Figure 6 below illustrates the required dependencies to bootstrap a new production environment and destroy an existing production environment.

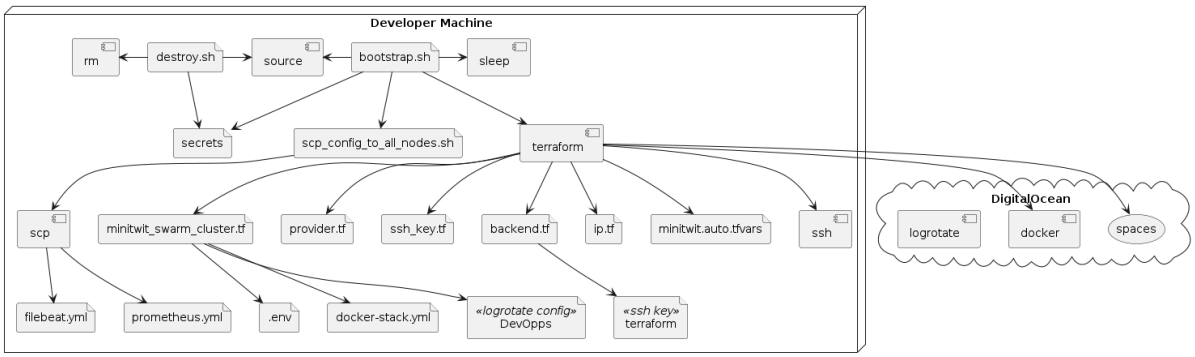

Figure 6: An illustration of the bootstrap and destroy script's dependencies. A full resolution image can be found here

Figure 7 below shows the dependency of the deploy GitHub actions that is used to deploy new versions to production.

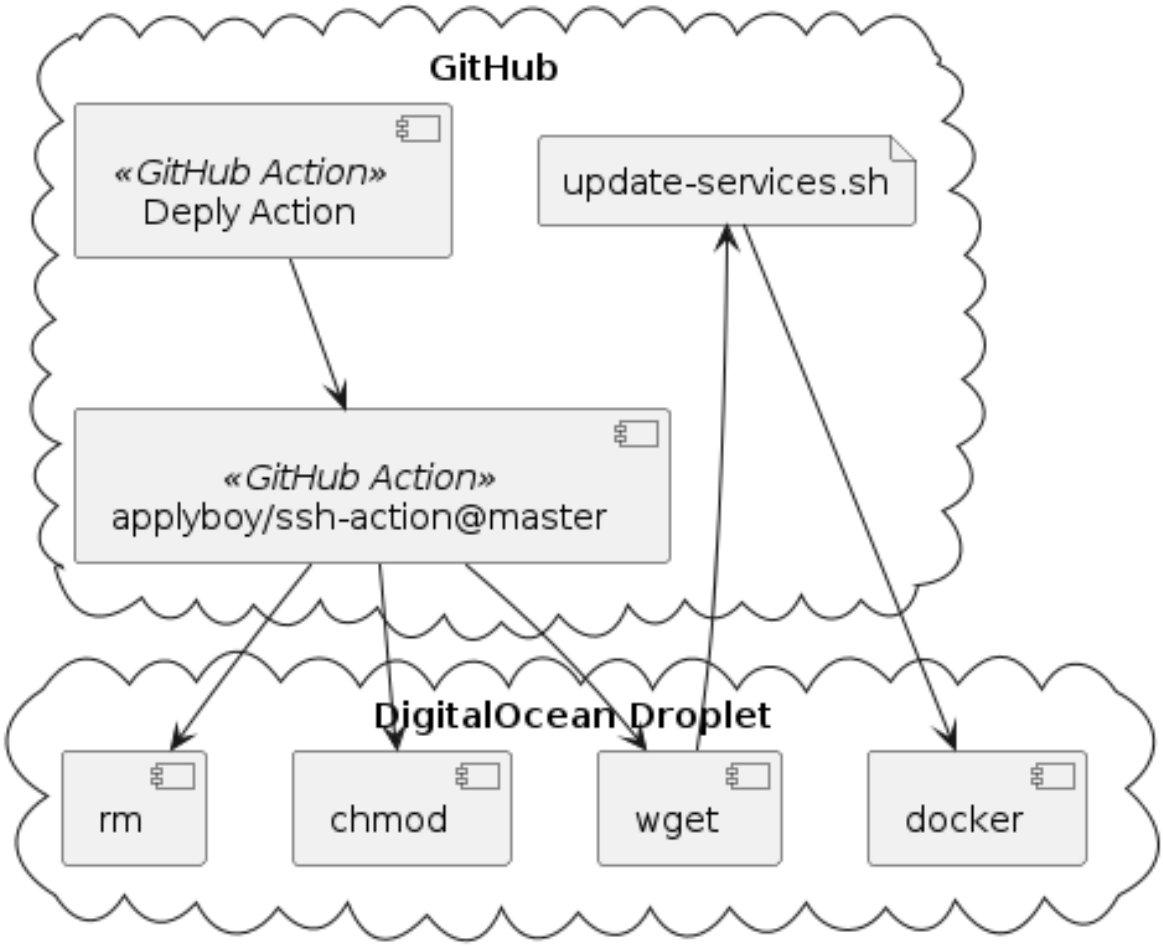

Figure 7: An illustration of the deployment dependencies

In addition to the *appleboy/ssh-action@master* GitHub action, the *Build and Deploy* action depends on several more actions to build the *Minitwit* system and package it using Docker containers. They are listed below:

- actions/checkout@v2
- bahmutov/npm-install@v1
- actions/upload-artifact@master
- actions/download-artifact@master
- docker/setup-qemu-action@v1
- docker/setup-buildx-action@v1
- docker/build-push-action@v2

Bootstrapping and destruction of production environments, as well as the *Build and Deploy* GitHub action depend on a number of Debian packages. Table 2 lists these packages and references the figure showing their dependency graphs.

Package	Dependency Graph
rm	No dependency info found
sleep	No dependency info found
terraform	No dependency info found
scp	No dependency info found
chmod	No dependency info found
ssh	Figure 8
docker	Figure 9
logrotate	Figure 10
wget	Figure 11

Table 2: Debian packages and the corresponding dependency tree

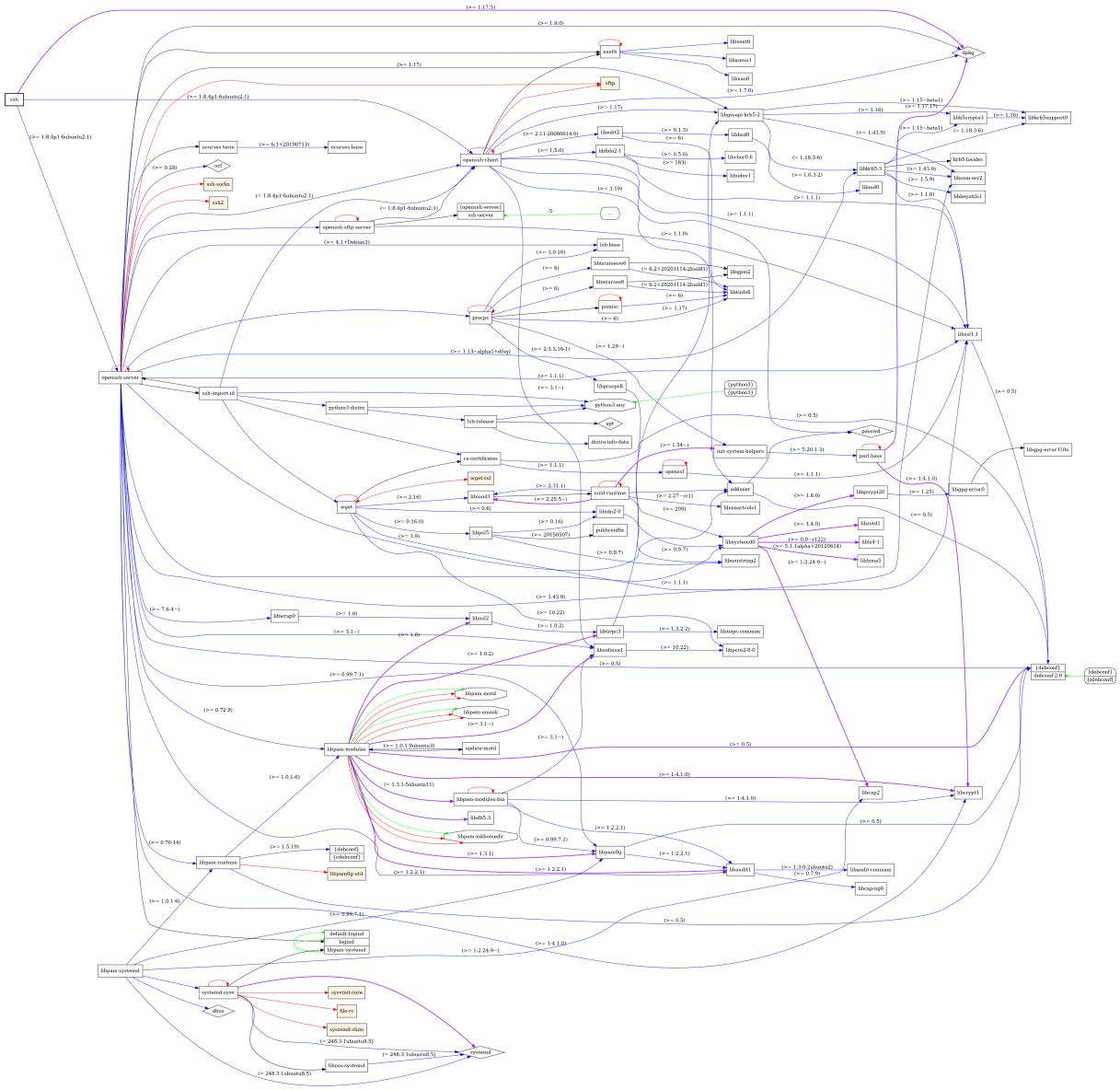

Figure 8: SSH's Dependency Tree. A full resolution image can be found here.

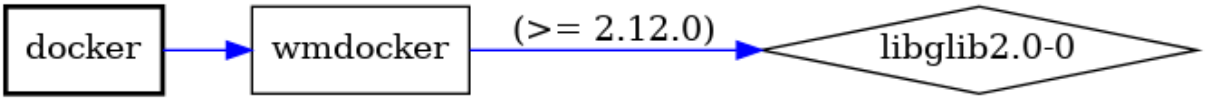

Figure 9: Docker's Dependency Tree

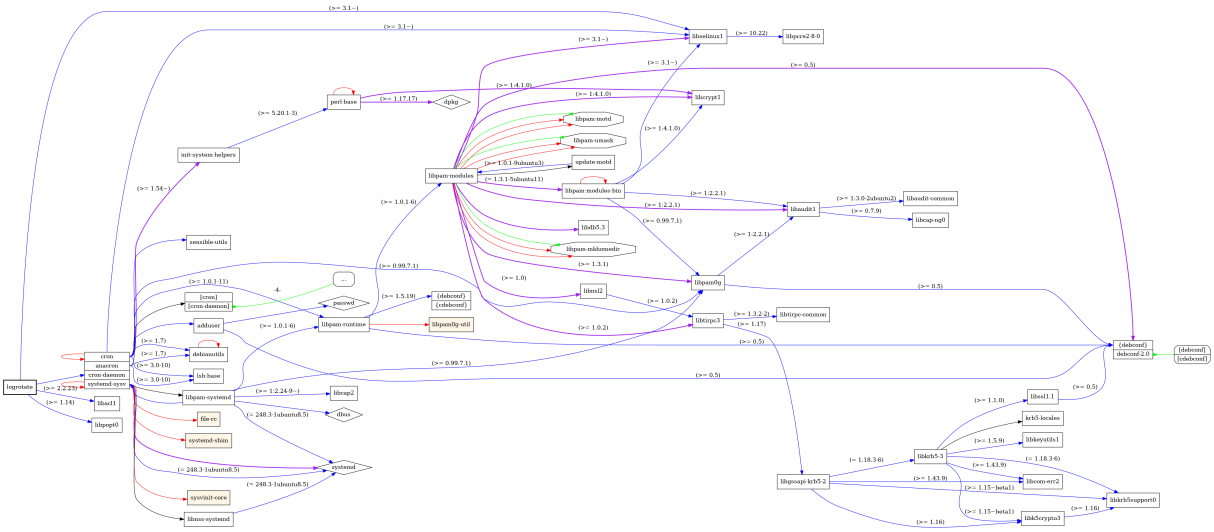

Figure 10: Logrotate's Dependency Tree. A full resolution can be found here here.

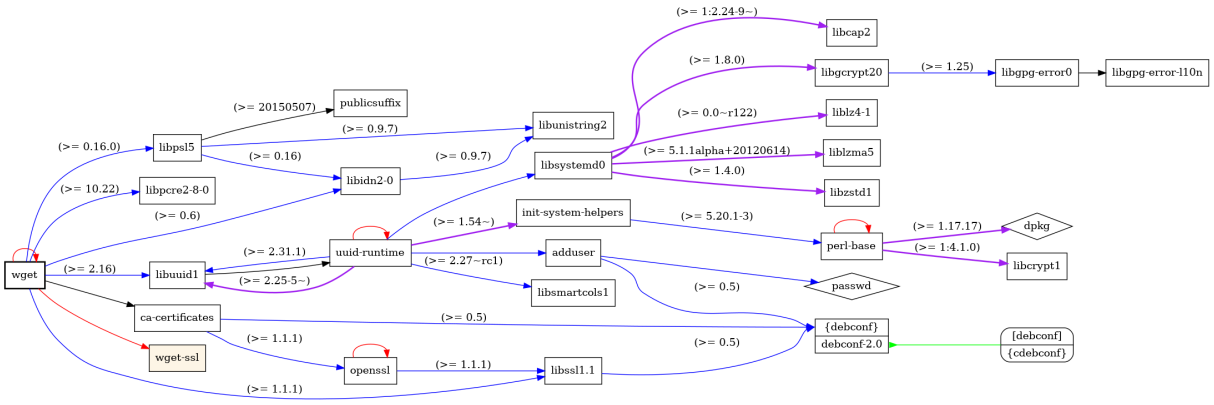

Figure 11: wget's Dependency Tree. A full resolution can be found here here.

C An Snapshot of the Project Board

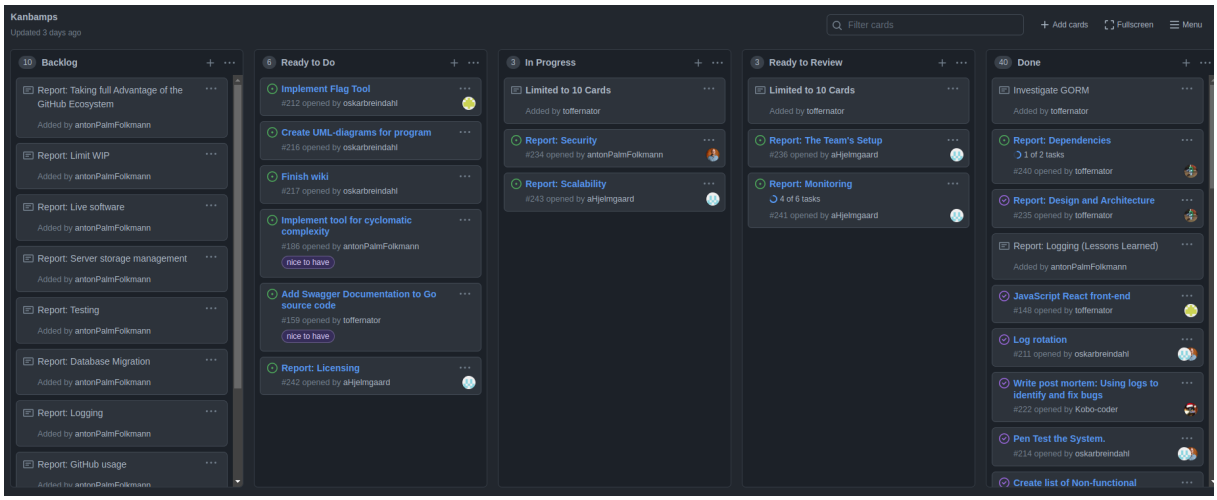

Figure 12: A snapshot of our Project Board towards the end. We restrict WIP using the "Limited to 10 cards" clarification. You can see the board live here