Summer 2018 Internship at Solvace

Simon Schellaert

Augustus 6 - September 14

Contents

1 Introduction

For six weeks in the summer of 2018, from Augustus 6 to September 14, I interned at Solvace. In this report, I'll elaborate on the company, my work there and how I experienced this period.

1.1 About Solvace

Solvace NV is a Ghent-based technology company that specializes in developing tools to aid problem solving in organizations. To that end, they offer an advanced web application that is eponymously called Solvace. This web app provides a comprehensive suite of business productivity tools that leverage machine learning to improve the performance of organizations.

The web app Solvace is a very diverse product with a wide range of features. One of the more prominent features is the problem solving view. This allows users to progressively break down problems in smaller problems. At each step of the process, Solvace provides contextual suggestions (i.e. structure, documents, expert knowledge, ...) based on similar problems that were previously solved. This allows organizations to tackle challenges more easily and more efficiently by allowing them to reutilize previous solutions. Every time someone solves a challenge, the whole organization thus becomes smarter through this growing knowledge base. This way, the accumulated knowledge of the entire company can be leveraged to make sure employees don't repeat past mistakes.

Another noteworthy feature is the meeting dashboard, which allows meetings to proceed as efficiently as possible. Among other things, it gives every participant access to all relevant information regarding the discussed topics. During the meeting, proposed actions can be added to the agenda to make sure they are carried through and monitored.

Finally, Solvace offers a so-called *toolkit*. In 2016, the company was acquired by EFESO to complement the latter's consulting offering. Conversely, Solvace could benefit from all the knowledge accumulated by EFESO by integrating this into their product. Specifically, EFESO possesses a huge toolkit consisting of thousands of documents in various formats, all linked to each other. This toolkit forms an exquisite collection of valuable knowledge. The Solvace application allows users to leverage this toolkit with ease by offering helpful directions based on related documents. Users can also cherry-pick pages from documents to conveniently aggregate all information relevant to the problem at hand. This part of the application is an indispensable tool that is used daily by the consultants at EFESO.

Even though Solvace has been acquired by EFESO, it still very much operates as an autonomous entity. The whole Solvace team is located at Dok Noord in Ghent, which minimizes communication overhead and enables fast iteration. The acquisition also allowed Solvace to grow more quickly, both in number of users and number of employees.

1.2 About the internship

As stated before, Solvace has been growing fast. This expansion poses some interesting scalability and workflow challenges, which can be alleviated by a more modern and flexible production environment than the one currently in use. Therefore, the goal of my internship was to turn Solvace

into a cloud-native application running on Kubernetes [1].

Up until this point the app had been developed under the assumption of running on a single monolithic server. This led to the application requiring a significant amount of non-trivial changes in order to become a good 'cloud-native citizen'. To allow me to gradually become acquainted with the massive code base, the internship was divided into a few work units that each accomplished a specific step towards the final goal. One example of such a work unit was, for example, to convert local file storage to cloud object storage or to get a minimal deployment running. In what follows, I'll elaborate upon each of these required steps and how I fulfilled them.

2 Technical report

2.1 Choosing a cloud provider

One of the first steps of running an application in the cloud is selecting a suitable cloud provider. Before the start of my internship, the offerings of a few major cloud providers were compared and Google Cloud Platform (GCP) was decided upon as the platform of choice [2]. One of the important factors behind this decision was the extensive and competitively priced range of services offered by GCP. Kubernetes, for example, is a rather complex system and is notorious for being difficult to set up and maintain. Rather than installing it ourselves, GCP allows us to take advantage of Google Kubernetes Engine (GKE). This is a production-ready Kubernetes cluster, managed by the experienced engineers at Google and configured with reliability, scalability and security in mind.

After analyzing the existing application, we determined that we were going to primarily rely on four services: the aforementioned GKE for compute, then Cloud SQL and Cloud Storage for data storage and finally Stackdriver for logging. One of my first tasks was acquainting myself with each of these services in order to set them up correctly for our purposes. Each of these services can be configured using the GCP command line tools or using a unified web interface called Google Cloud Console. In order to make each configuration step clear and reproducible, I wrote extensive documentation that lists the reasoning behind each configuration step and the corresponding command to execute that step. This makes it possible to start a new GCP project from scratch and fine-tune it to our needs in a matter of minutes.

2.2 Getting a minimal version of the application running

After setting up the cloud environment to run the application, the next step was gradually migrating the application itself to GCP. As mentioned before, Solvace is a rather complex application with lots of moving parts. Luckily, however, the application is able to function, albeit with some key features disabled, in a simple environment consisting of only a Rails app server and a MySQL database. This allows developers to easily get a basic version running on their local machines with minimal set-up. It thus made sense to first attempt to get this basic version running on GCP before introducing more complex components.

I started with setting up a Kubernetes Deployment consisting of two containers: a Rails container running the app itself and a so-called proxy sidecar container, used by the main container to connect to the Cloud SQL database. The infrastructure was already in place to automatically build a Docker

container and push it to a private container registry for each commit to `master`. So, I was able to use these containers as a starting point and adapt them so they could run on the Kubernetes cluster. Most of the modifications at this point were rather simple, but they allowed me to get acquainted with some key parts of the code base. The most noteworthy modifications were the following:

- **Storing all configuration in the environment**
  The application already used environment variables to store most configuration options, such as database credentials. A few configuration options, however, were still hardcoded in Ruby, while they had to be assigned different values when running on Kubernetes. So I extracted these options and allowed them to be filled in using environment variables, as recommended by the Twelve-Factor App methodology [3].

- **Serving a 200 OK status code on the root path**
  The GCP load balancing health check expects the web application to return a 200 OK status code on its root path. In our case, a 302 Moved Temporarily status, redirecting to an error page, was returned since the request made by the health check doesn't include the `Host` header used by the app to distinguish between different clients. This was solved by adding a Rails `before_action` filter to check if the `User-Agent` indicates a request made by the health check and not redirecting the user in that case.

- **Working around a OverlayFS kernel driver bug**
  When running the application the first time on Kubernetes, I noticed that the application logs were not printed to standard output. Further research indicated that this was caused by a long-standing bug in the OverlayFS kernel driver [4]. This bug prevented `tail -F` from following a file that is copied up to a higher layer. Luckily, the workaround was rather simple: I removed the affected file (`production.log`) in the base image to make sure the file isn't copied up to a higher layer when we run the container. Because of this change, te file is directly created in the uppermost layer and `tail` can correctly track it.

Note that these modifications were all implemented in a backward compatible manner and pushed to `master` after extensive testing. That way, the same Docker containers could run both on Kubernetes and on the current infrastructure.

Finally, I allocated a static IP address and created a Kubernetes Service associated with that address to expose this preliminary Deployment to the outer world. This allowed me to log in to the application and verify that everything was working as expected.

2.3 Creating a Helm Chart

Having a basic version of the app up and running, the next step was to get some more advanced services working to get a feature-complete version of the app. I started off by creating plain Kubernetes YAML configuration files for each Kubernetes object (Deployment, Service, StatefulSet, ...). Frequently, these files needed some external configuration (e.g. the knowledge of the environment being *staging* or *production*) while still sharing most of the configuration. I initially used placeholders for these values in the configuration files, that were then filled in using GNU's `envsubst`. It, however, became quickly clear that this ad-hoc approach was neither flexible nor easily maintainable.

A more structured and elegant solution for specifying Kubernetes deployments is provided by Helm, the Kubernetes package manager [5]. Helm allows users to easily templatize their configuration files using the Go template language. This makes the configuration files both more readable and flexible. Furthermore, Helm also allows distributing the whole configuration in a so-called Chart (analogous to a `deb` or `rpm` package). This provides a maintainable and user-friendly installation option for complex Kubernetes apps in an existing cluster. While the usage of Helm does introduce an extra layer of abstraction, it's rather simple to get started with and a lot more maintainable in the long run than any ad-hoc solution. We thus decided to fully embrace Helm and create a private Chart for Solvace, complying with all the best practices set forth by the Helm community.

After some rounds of incremental improvement, I arrived at a Chart consisting of the following main components:

- **Main Ingress**
  This is an Ingress resource that routes incoming HTTP(S) and WebSocket traffic to the back-end services. On GCP, Ingress resources are backed by an L7 load balancer that automatically monitors the back-end services and routes only to correctly-functioning services. The load balancer is also equipped with a set of forwarding rules based on the requested path. In our case, the rules are simple: All traffic whose path starts with the substring `/faye` is routed to Pods running Faye and all other traffic is routed to Pods running Rails (see below).

- **Faye**
  This component consists of two resources: The first is a Deployment of a Faye WebSocket server using the Private Pub gem, allowing the app to provide real-time updates through an open socket without tying up a Rails process. The second is a Service exposing this server to the aforementioned load balancer. Together, these resources form the driving force behind the real-time collaborative editing functionality.

- **Memcached**
  To speed up the application, the Rails application uses the Dalli gem to cache the result of calculations on a Memcached server. This component too consists of a Deployment and a corresponding Service exposing it to the Rails app. The only particularity here is a custom health check that periodically tries to establish a TCP connection with the Memcached server. If the server fails to pass a number of successive health checks, it is automatically restarted.

- **Rails**
  This is essentially the minimal version of the app as described in Section 2.2. Once again, it consists of both a Deployment and a Service to make it available to the load balancer. As stated before, a Cloud SQL proxy sidecar runs alongside the main container to allow it to connect to the database. The image for the main container is provided by Solvace's private container registry and runs the Phusion Passenger app server.

- **Rake Tasks**
  This component consists of a single Job. It differs from the other components in the sense that it doesn't provide a Service to connect to nor is it always running. The Job is attached to a *pre-upgrade hook* to make it run automatically each time the application is updated. The Rake tasks are responsible for syncing the assets and performing any database migrations.

- **Sidekiq - Main**
  The application heavily relies on background processing for both core functionality and maintenance tasks. This background processing is provided by a Sidekiq Deployment. The workloads themselves are communicated to this component using Redis. Since the Redis component is set up to persist data (see below), scheduled workloads don't get lost if a Pod restarts.

- **Sidekiq - Toolkit**
  Syncing the toolkit (see below) is a special kind of periodic workload since it tends to take a lot longer and uses a lot more resources than other tasks. By having a Sidekiq Deployment specifically aimed at running the toolkit synchronization, we can dedicate extra resources (e.g. more CPU cores) to this workload. The configuration of this component is very similar to the *Sidekiq - Main* component above. The major difference is that this Deployment has more resources dedicated to it and only accepts jobs from the `toolkit` task queue.

- **Redis**
  This component consists of a StatefulSet running an official Redis image and a Service exposing it. We are using a StatefulSet instead of a Deployment since we need to attach a persistent ReadWriteOnce volume to make sure Redis data is always persisted on disk and never lost. While it is technically possible to attach a ReadWriteOnce volume to a Deployment, doing so is not recommended since it may result in a deadlock in some circumstances. It's important that the Redis data is persisted on disk to make sure that queued tasks, like sending notification emails, don't get lost upon a restart.

- **Cloud SQL proxy**
  The attentive reader may have noticed that some of the components above (Rake Tasks, Sidekiq, ...) need access to the database but don't have an explicit Cloud SQL sidecar container listed. Instead, these containers all connect to this proxy to access the database. The reason for this decision is two-fold: First, a Pod only terminates if each container in it terminates. A Cloud SQL sidecar container, however, never terminates. This means that Jobs (like Rake Tasks) would hang around forever [6]. Using a centralized Cloud SQL proxy that the Jobs connect to, solves this issue. Second, the components listed above don't heavily utilize the database. So it makes sense for them to share a single Cloud SQL proxy.

Each of the components listed above is declaratively described in one or more YAML configuration files using the Helm template syntax. This allows easily spinning up a staging or production environment by passing the appropriate flag to the `helm install` command.

2.4 Migrating toolkit to cloud storage

As mentioned before, the toolkit built into the application is a crucial tool that is extensively used by the consultants at EFESO. The contents of this toolkit are maintained by a dedicated team at the EFESO headquarters and the latest changes are synced to a production server of Solvace multiple times a day. On completion of this synchronization job, a Rake task called `fill_toolkit` is run. The job of this task is to prepare the raw toolkit data for use in the application by fulfilling the following functions:

- **Converting legacy file types to their modern equivalent**
  For example, converting the older PPT format to the new PPTX format. By doing this

conversion, we only have to concern ourselves with the newer Office Open XML formats from this point onward and we don't need to make subsequent operations compatible with the older binary formats.

- **Indexing the contents of each document to enable full-text search**
  For example, the contents of each slide of a Powerpoint presentation are extracted and saved in the database. This enables full-text search and contextual suggestions.
- **Generating thumbnails for each document**
  For example, the slides of a PowerPoint presentation are extracted and converted to images. This enables fast previews of specific pages of documents at various places in the application.

After these steps, the latest version of the toolkit is ready for use in the application. Each time a user attempts to access a toolkit file, the rights assigned to that user are checked by the Rails application. First, we make sure that the user has the necessary permissions to access the requested file. Next, we check if the user has previously edited the content of the requested document (e.g. changed the titles of slides). In case the user did make changes, we obtain the relevant modifications from the database and apply these to a copy of the original toolkit document before serving it to the user. If the user did not make any changes, we can simply serve the original toolkit file. In both cases, Rails instructs Nginx to serve the file using the `X-Accel-Redirect` response header as to not tie up the Rails process.

While this approach works and is relatively simple, it has some drawbacks regarding scalability and flexibility. A first drawback is that we are responsible ourselves for storing and serving each toolkit file. As noted above, the toolkit is heavily used and the cost of serving it is significant. There are specific cloud storage providers out there, such as Google Cloud Storage (GCS), that can serve static files both faster and cheaper than we can. Another drawback is that all of the toolkit files (which amount to a few hundreds of GBs) have to be accessible both to the Nginx pod and the Rails pod, which is non-trivial in Kubernetes. We thus opt to store the toolkit on GCS and discuss the details of this new setup next.

2.4.1 Syncing the toolkit

As noted before, the toolkit is composed of two data sets. The first data set is the raw data that is coming directly from the EFESO headquarters. This data is then processed using the `fill_toolkit` Rake task and stored in a location accessible to end-users (provided they have the necessary rights). This processed data then forms the second data set. Currently, a cron job at EFESO periodically executes an `rsync` to the Solvace servers, which detect when the rsync is finished and then start the Rake task. If we store the toolkit on GCS, however, we're no longer able to listen for this event. We solve this problem by extending the application with two authenticated HTTPS endpoints: `/toolkit/start` and `/toolkit/stop`. These endpoints expose a mutex that is also used by the Rake task. Periodically, a cron job at EFESO attempts to acquire the mutex by calling `/toolkit/start`. If this fails (i.e. returns a non-200 OK status code), this means the Rake task is running and we postpone the synchronization operation initiated by EFESO. If this succeeds, however, we are sure that we won't interfere with the Rake task and thus start the synchronization from EFESO to GCS using the *gcloud* utility. Once the synchronization is finished, `/toolkit/stop` is called, which releases the lock and schedules the Rake task. This strategy ensures that the toolkit synchronization and Rake task can never run at the same time and interfere with each other.

2.4.2 Converting raw toolkit data

The existing Rake task heavily relied on the assumption that both the source (raw data) and destination (prepared data) reside on the local file system. For example, in order to avoid converting unmodified files, a SHA-1 hash of each source file is computed and checked against a list containing hashes of files we previously converted. Now, retrieving the SHA-1 hash of a local file is cheap. Retrieving the SHA-1 hash of a file stored on GCS, however, is very expensive since the whole source file has to be downloaded before its hash can be computed.

Another example of this reliance on the local filesystem, is the myriad of calls to `Dir.exist?()` to check if a directory with the specified name exists in the source data. These calls are necessary to derive the correct path for a file listed in the index. On GCS, however, the very concept of directories is absent with forward slashes in the name of an object not having any special meaning. Due to these issues, a significant portion of this complex Rake task had to be rewritten. Specifically, I rewrote the task to make use of an abstract class `FileProvider` which then has subclasses for retrieving and storing data either on the local filesystem or on GCS. By doing so, the task doesn't have to be aware whether local file storage, cloud object storage or any other storage mechanism is used. Most of the needed functionality (e.g. retrieving a file with a specific name) could be straightforwardly translated to calls to the GCS API. However, the two issues outlined above (usage of SHA-1 hashes and absence of directories) remained problematic.

To fix the first issue, we migrated from SHA-1 hashes to MD5 hashes, since MD5 hashes are provided by GCS as part of the object metadata. That way, we don't have to download the whole file first and calculate its hash ourselves to check whether it has changed. While this change was rather easy in terms of code, it also meant that we had to migrate the production database to store the MD5 instead of SHA-1 for all previously imported files. This conversion was more involved but was ultimately successfully completed and deployed.

To fix the second issue, we opted to let the GCS file provider mimic directory functionality. We chose this option since modifying the task to not rely on the availability of directory functionallity turned out to be difficult. Concretely, a list of all objects in the bucket is retrieved and an in-memory file system tree is then constructed based on slashes in object names. This tree can then be used to efficiently query the existence and contents of specific directories.

Finally, we also implemented a `copy()` method in the GCS file provider that utilizes the copy functionality in the Google Cloud SDK. This method allows copying files between buckets (i.e. the bucket containing raw toolkit files and the bucket containing prepared toolkit files) without having to download and re-upload them. Doing so drastically speeds up the whole procedure since a significant portion of the files can be copied without needing any conversion.

2.4.3 Redirecting users to Google Cloud Storage

After the preparation steps above, both raw and prepared toolkit files are now located in a GCS bucket and are ready to be served to the users. However, an important feature we didn't take into account yet is the ability of users to privately edit the content of documents in the toolkit. These changes are then stored in the database in a declarative format. Before the migration to GCS, if the user requested a toolkit document, the back-end checked if this document had been

edited by the current user and then either located the unmodified file or applied the modifications to a copy of the stored document. In either case, the resulting file is served by Nginx by passing the `X-Accel-Redirect` response header. Note that the overwhelming majority of the requests are for unmodified documents. The scenario in which the document is modified occurs only infrequently.

The approach above could by adapted by redirecting the user to GCS instead of using the `X-Accel-Redirect` response header in case the file was not modified (or downloading from GCS, modifying and serving the file ourselves in case it was modified). This, however, is not ideal since every request would then first hit our own back-end before being redirected, resulting in a doubling of the latency experienced by the end user (Figure 1). To solve this issue, the client needs to know whether he can find the file on GCS or not *before* the actual download call. We thus modify the listing endpoint `/tk_analyses` to return a `view_url` for unmodified documents. That way, the client can fetch unmodified documents directly from GCS without any interaction on our part and only has to contact our own back-end if the file is not available there (Figure 2). Since only modified files are not available on GCS, most of the files will be served directly from GCS without having to contact our servers first.

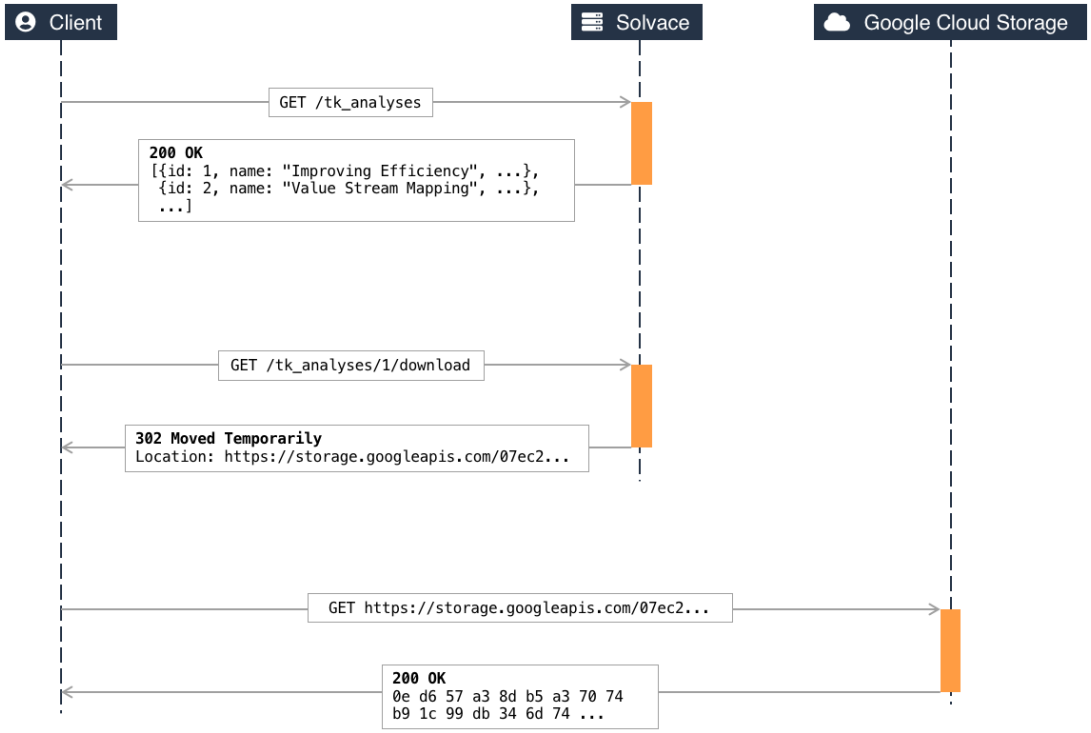

Figure 1: Naive approach: redirecting the user upon item retrieval

2.4.4 Generating signed URLs

We can't simply pass links to toolkit files on GCS to the client since the used bucket is not public, nor do we want it to be due to confidentiality considerations. Furthermore, users typically only

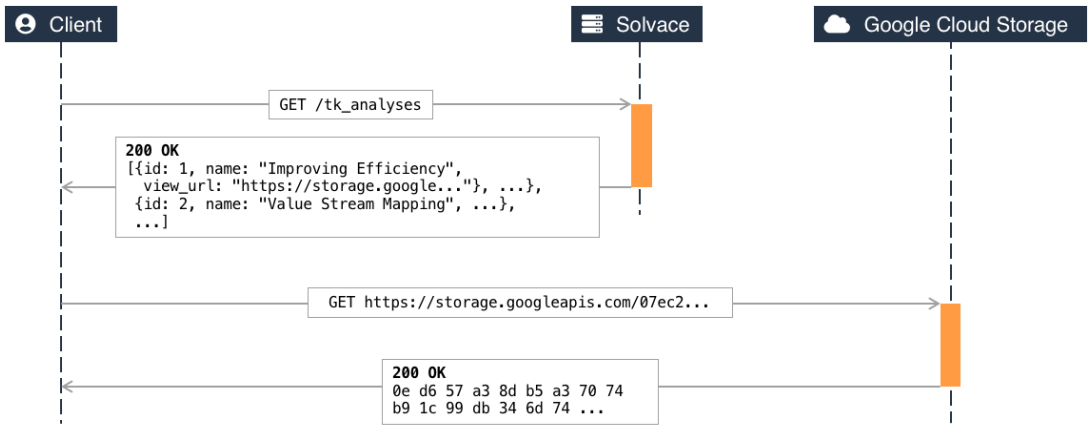

Figure 2: Improved approach: sending the signed URL beforehand

have access to a limited subset of the toolkit and this subset may change over time. It would be unacceptable to have users accessing objects on which their access has been revoked, or worse, non-users guessing the names of objects in the bucket to access confidential documents. We thus make all objects private and explicitly provide access to authorized users only using signed URLs, a mechanism for query string authentication.

Specifically, each time we need to pass a link to a GCS object to a user, we generate a signed URL using our secret key. That way, we can give time-limited read access to users without having to fetch the file ourselves. That signed URL is an HMAC over the plain URL to the object concatenated with an expiration timestamp. The resulting hash is then appended to the plain URL as a query parameter to form the signed URL. To be able to immediately revoke the access of users to specific toolkit files, we would like the signed URL to expire as soon as possible. There is, however, a trade-off here since providing users with new authentication query parameters causes any copies of the object in their local cache to become irrelevant. Ultimately, setting the expiry date of all URLs to the next occurring Sunday noon was deemed a good trade-off between cacheability and security. Using this schema, accessing the same file multiple times in a given week, results in hitting the browser's cache, while access revocations will be established by the end of the week. Note that setting this expiration date only affects files for which the user already obtained a signed URL. Of course, we don't sign any more URLs for a user once his access rights have been revoked.

2.4.5 Providing developers with access to the toolkit

A nice side effect of the changes above is that each developer on the team can now easily get access to the full toolkit on his local development machine. Previously, the Rails application expected the toolkit to reside on the local file system. Since the toolkit takes up a huge amount of space, developers had to resort to either testing changes locally with only a small fraction of the toolkit or to testing in the staging environment, where debugger support is limited. Now that the Rails application is able to serve the toolkit from GCS, we can easily give each developer access to a full local toolkit installation. For that purpose, an anonymized snapshot of the toolkit database is

provided to the developers. Once they import this snapshot, which is only a few megabytes, they can browse and experiment with the full toolkit on their development machine. The files themselves are then stored in a GCS bucket. This significantly reduces the friction between noticing a bug in the toolkit component and fixing it.

2.5 Setting up continuous delivery

The previous steps resulted in a full-fledged deployment of Solvace on Google Cloud Platform. Of course, setting up a working version of the application is only a small step in modern software delivery. Solvace aims to frequently release reliable and high-quality updates of their application to their users. This goal is realized partly through a principle called continuous delivery (CD). Specifically, continuous delivery means that the code on `master` is always ready to go into production at the touch of a single button.

A significant part of the work needed to set up CD for the new production environment was already done since this methodology was already being practiced. Specifically, GitLab CI/CD is being used to automatically run tests and create a Docker image for each commit to master. A button was available in the GitLab UI to deploy the latest commit on the master branch to production using an Ansible playbook, as detailed in the file `.gitlab_ci.yml`. Setting up CD for Kubernetes involved declaring a new job in this same file. The new job runs the following sequence of instructions in a fresh container based on the `google/cloud-sdk` image:

1. Authenticate with the credentials provided as environment variables using the `gcloud` utility.
2. Obtain a local copy of the latest version of the Helm Chart.
3. Upgrade the installation based on the Chart and the `CI_COMMIT_SHA` environment variable.
4. Print the logs of the migration tasks.

In an analogous manner, it is also possible to manually roll back to a previous deployment by clicking a single button next to the desired commit in the GitLab UI. Apart from this manual deployment to production, each commit to `master` is automatically deployed to a staging environment that matches the production environment as closely as possible.

2.6 Setting up centralized logging

The final step towards a production-level Kubernetes cluster is setting up logging and monitoring. To this end, Google Cloud Platform offers Stackdriver [7], a fully managed service that integrates with the other GCP services we're already using. By default, Stackdriver aggregates the standard output and standard error output of the pods running on Google Kubernetes Engine. Since logs were already printed to these streams before migrating the application to GKE, the app didn't need to be modified to enable log aggregation.

Since most other GCP services (e.g. the HTTP(S) load balancers that back Kubernetes Ingress resources) are readily integrated into Stackdriver, it is easy to set up monitoring for most key metrics. Examples include the percentage of `200 OK` responses or the 95 % response time. Based on this data, we set up a comprehensive dashboard to visualize important key metrics and enabled

alerts to get notified when these significantly deviate from their typical values. That way, a quick glance suffices to get an overview of the status of all components and any anomalies are reported immediately.

3 Evaluation

3.1 Impact on personal development

My internship at Solvace has been a great learning experience, both on a personal and professional level. Before starting my internship, I had only a vague idea of what software development actually looked like in practice. It was thus very interesting to be part of a real company for a few weeks and see how things are actually done over there. One of the aspects I particularly liked about Solvace is the fact that their engineering team is relatively small and very agile. This means that each individual person has a significant impact on the final product, which was a very fulfilling experience.

My time at Solvace will, undoubtedly, also prove indispensable when looking for a job. On the one hand, this experience has helped me to better understand what I really look for in a job. On the other hand, an internship is a great asset to put on your resume and will certainly make it more attractive to potential future employers.

And last but definitely not least, I also got to know some incredible colleagues. I would like to thank the whole team at Solvace, with Reinout and Jasper in particular, for providing me with this opportunity. They were not only keen to answer all my questions and help me whenever possible, but also made sure I felt right at home from the very first day. It was nice that I could turn to them for both interesting technical discussions and for some pleasant conversation during lunch.

3.2 Educational relevance

Based on my own experience, I would consider the internship an indispensable part of the curriculum. Next to the personal aspects that I focused on in the previous section, my internship also significantly increased my technical skills. Besides presenting an opportunity to put into practice a lot of what I learned in class, I also got the chance to work with some new exciting technologies. Finally, even though there are already a lot of project-based courses in the curriculum, these still don't exactly simulate the dynamics of working in a professional environment on an actual product. So, getting this real-world experience nicely complemented the other more theoretical courses.

4 References

[1] Kubernetes. `https://kubernetes.io/`.

[2] Google Cloud Platform. `https://cloud.google.com/`.

[3] The Twelve-Factor App. `https://12factor.net`.

[4] OverlayFS does not implement inotify interfaces correctly. `https://bugs.launchpad.net/ubuntu/+source/linux/+bug/882147`.

[5] Helm. https://helm.sh/.

[6] Kubernetes on GKE with CloudSQL proxy, and Cron Jobs. https://ihaveabackup.net/article/kubernetes-on-gke-with-cloudsql-proxy.

[7] Google Stackdriver. https://cloud.google.com/stackdriver/.