PROJET DE MASTER 2020

SCIENCES ET INGÉNIERIE DES RÉSEAUX, DE L'INTERNET ET DES SYSTÈMES

Plateforme d'éducation en ligne à l'échelle nationale

Auteurs
Paul HENG - Robin BISCAY - Hussein TAHER - Oussema TOURKI - Alexandre DOROFEEV

Encadrants
Pascal MERINDOL - Stéphane CATELOIN

Jury
Cristel PELSSER - Quentin BRAMAS

AUTOMNE 2020

Table des matières

1 Partie technique

1.1 Présentation générale du projet

La plateforme développée se positionne comme éducative. Elle a pour vocation d'une part de stocker des cours sous forme de texte (écrit au format Markdown) ainsi que des fichiers annexes (PDF, vidéos, images, etc ...) et d'offrir aux utilisateurs la possibilité d'accéder à ces cours et de les regarder sur la plateforme. Les utilisateurs connectés pourront discuter entre eux sur un espace forum divisé en branches associés aux différents cours présents. La plateforme est disponible sous forme d'un site internet accessible à partir des smartphones, PC, tablettes. Pour une meilleure visibilité, une notation sur 100 (assurée par les utilisateurs) est associée à chaque cours.

Le défi technique principal de la plateforme est de maximiser la qualité de service pour l'utilisateur en lui offrant une latence d'utilisation de la plateforme minimale peut importe sa position géographique dans l'hexagone. De même, la plateforme est pensée pour être distribuée, non seulement pour les performances, mais également afin d'offrir de la redondance lors de l'accès aux données, de sorte à ce que la perte d'un site n'empêche pas la plateforme de fonctionner.

La finalité du projet est d'offrir un mix entre le Moodle et un YouTube dédié uniquement à l'éducation qui sera lisible, intuitif et efficace.

1.2 Justification des choix techniques

1.2.1 Application

Nous avons choisi d'utiliser Django[1] pour écrire le serveur de l'application. Ce framework python contient de nombreuses fonctionnalités qui favorisent un cycle de développement rapide. Il simplifie la gestion de la création de pages statiques, la pagination, les listes d'objets, les systèmes d'authentification, la création des tables SQL de l'application et la rédaction de la documentation de l'API.

Notre serveur s'appuie sur un module nommé Django REST Framework, qui facilite la conception et la maintenance d'une API REST. Notre serveur est basé sur une architecture MTV (model-template-view), qui est une dérive de l'architecture MVC (model-view-controller), qui est le principal motif utilisé par les applications web.

Concernant la partie client, suite aux retours que nous avons pu obtenir de la part de différents élèves à l'Université, via un formulaire traitant de la solution actuellement en place (Moodle), une critique fréquente était le design de l'outil et son interface.

Pour tenir compte de ces retours, nous avons choisi de développer une "Single Page Application" (SPA) grâce au framework React[2]. Ce choix a principalement été motivé par la volonté d'offrir une expérience agréable à l'utilisateur, en proposant une interface dynamique avec une navigation qui ne nécessite pas de rechargement de page, toute la gestion du DOM étant déléguée à JavaScript.

1.2.2 Infrastructure

Pour s'adapter aux multiples contraintes liées au projet, et de par notre sujet, nous avons opté pour des solutions techniques avec une approche distribuée, permettant ainsi d'apporter de la haute-disponibilité, à l'échelle d'un site ou d'une région. De plus, nous avons également effectué ces choix en ayant pour objectif la scalabilité horizontale de la solution.

Orchestration de conteneurs avec Kubernetes Dans l'optique de fournir la meilleure qualité de services possible, nous avons opté pour Kubernetes[3]. Cette solution d'orchestration dispose de nombreux outils permettant d'adapter et de répartir la charge lors des accès à l'application.

Déployé en haute-disponibilité (3 noeuds de contrôle et 3 noeuds de calcul), Kubernetes nous a permis de déployer un nombre de conteneurs minimal sur l'infrastructure, tout en permettant un redimensionnement rapide en augmentant le nombre de répliques (qui sont créées rapidement, étant donné qu'il s'agit de conteneurs et non de machines virtuelles). De même, si un noeud vient à tomber, les conteneurs présents sur le noeud défaillant sont automatiquement recréés sur les noeuds encore en service, afin de limiter l'impact sur les performances en cas de panne.

Nous avons pensé notre projet avec en tête la scalabilité horizontale, et un orchestrateur de conteneurs est idéal dans ce cas de figure, car il suffit simplement de configurer les nouveaux noeuds pour rejoindre le cluster existant pour que les différents services soient automatiquement propagés.

Enfin, l'utilisation de Kubernetes, qui se fait principalement avec une approche déclarative sous la forme de fichiers YAML, permet d'adopter une démarche de type Infrastructure-as-Code, ce qui permet une meilleure gestion de l'infrastructure ainsi qu'un déploiement facilité si le projet vient à s'étendre sur de nouveaux sites.

Stockage objet distribué avec Ceph La haute-disponibilité étant l'élément central en ce qui concerne la motivation des choix techniques pour l'infrastructure, nous avons opté pour une solution de stockage distribuée grâce à Ceph. Comme nous avons utilisé Kubernetes, nous nous sommes basés sur le projet Rook[4], qui permet de déployer un cluster Ceph au sein même de Kubernetes. Nous avons utilisé Rook / Ceph pour le stockage objet, qui est totalement compatible avec l'API S3 d'Amazon (qui est le standard le plus utilisé actuellement). Déployer Ceph au sein de Kubernetes nous a permis d'utiliser les fonctionnalités de Kubernetes en parallèle (répartition de charge automatique entre les différents noeuds, gestion automatique de la terminaison TLS).

Enfin, étant donné que l'objectif était de déployer notre solution sur plusieurs sites, nous avons mis en place une réplication du stockage objet entre nos 2 sites. La réplication étant basée sur un modèle actif / actif pour les lectures, cela permet également d'améliorer le temps de réponse et les débits en ayant la possibilité d'utiliser l'instance S3 la plus proche de son emplacement.

Base de données distribuée avec CockroachDB Notre projet étant pensé pour être utilisé à l'échelle nationale, nous avons essayé de proposer un service avec un modèle actif / actif sur les différents sites, dans l'optique de pouvoir récupérer l'ensemble des ressources depuis le site le plus proche de la localisation de l'utilisateur. Pour la base de données, nous avons d'abord pensé utiliser une base de données traditionnelle, PostgreSQL en l'occurrence, mais ce type de solution est pensé pour une scalabilité verticale, ce qui n'est pas notre approche. Après avoir considéré une éventuelle solution de type NoSQL (MongoDB), nous avons eu l'occasion de découvrir les solutions appelées NewSQL, qui sont des bases de données naturellement distribuées et relationnelles. Après avoir eu l'accord du client, nous avons loué des ressources chez un hébergeur en ligne pour disposer d'un troisième site pour cette preuve de concept, nous avons déployé CockroachDB[5]. Le troisième site était nécessaire afin de garantir une haute-disponibilité à l'échelle régionale de la base de données. En effet, cette solution utilisant Raft[6] comme algorithme pour le consensus (à l'instar de Kubernetes), il était nécessaire de disposer de 3 sites afin de respecter le quorum et de tolérer la perte d'un site entier. CockroachDB a été déployé sur 3 noeuds par site, afin d'obtenir un consensus à la fois à l'échelle locale (3 noeuds), et à l'échelle régionale (3 sites). Cette solution présente plusieurs avantages, dont une étroite compatibilité avec PostgreSQL, ce qui a permis de récupérer l'existant lors de la migration vers cette nouvelle technologie. Les interactions avec le SGBD se font via une interface SQL traditionnelle, mais les données sont ensuite sérialisées et la logique interne de l'outil est gérée grâce à un stockage clé-valeur.

Pour pallier les latences introduites par les communications entre les différents sites, des fonctions de partitionnement et de réplication des données existent au sein de la solution.

1.3 Architecture matérielle déployée

Comme évoqué dans la section 1.2.2, la haute-disponibilité et l'approche distribuée sont les éléments sur lesquels nous nous sommes le plus focalisé.

1.3.1 Site C315

La figure 1 présente le déploiement effectué en salle C315, pour notre site appelé "SXB". Comme la plupart de nos technologies se basent sur une architecture à 3 noeuds (donc tolérant une défaillance), nous avons réparti les différents éléments à raison d'un part noeud.

Sur chaque noeud, nous disposons donc de deux VM dédiées à Kubernetes (un noeud maître et un noeud de calcul). Une instance CockroachDB est également présente sur chaque noeud, et dispose de son propre volume de stockage.

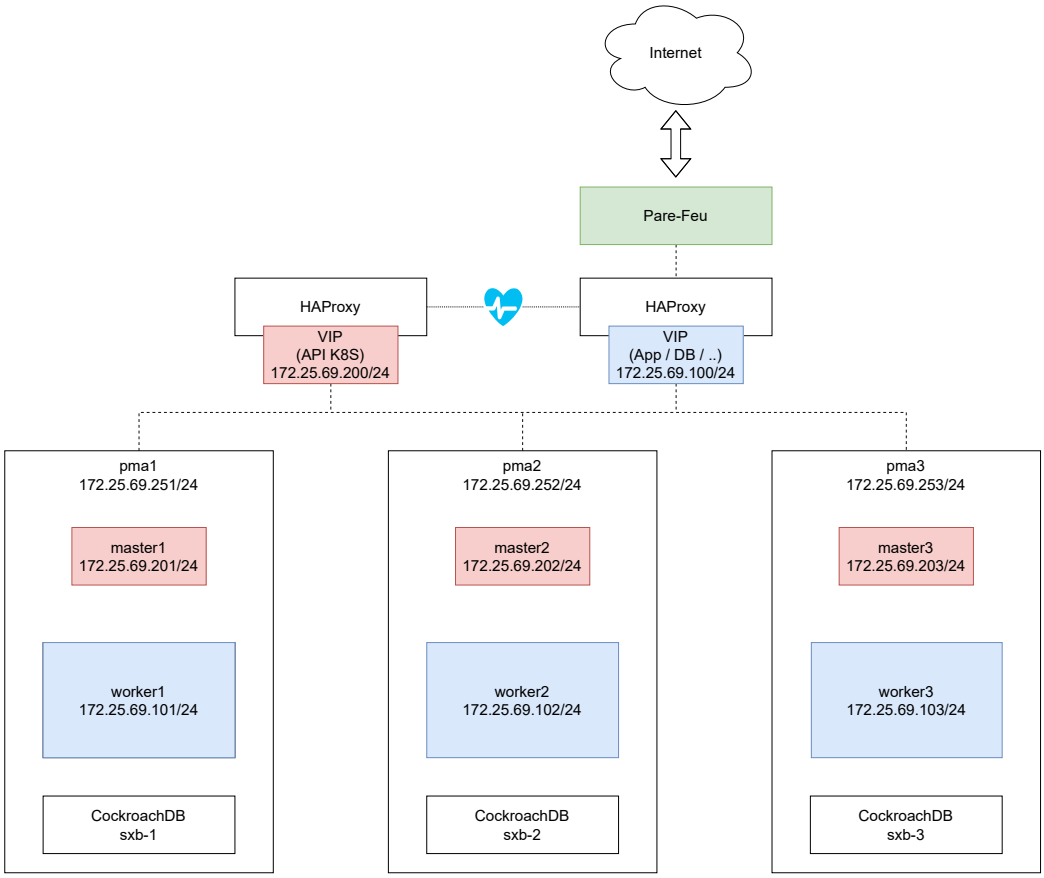

FIGURE 1 – Déploiement sur le site SXB (C315)

Afin d'assurer la haute-disponibilité et de répartir la charge entre les différents noeuds, nous avons mis en place deux instances HAProxy[7] (sur des Raspberry Pi).

Afin d'assurer la redondance au niveau des répartiteurs de charge, ces derniers se partagent des IP virtuelles grâce à l'outil Keepalived[8], qui utilise VRRP. Par défaut, un répartiteur est dédié au trafic applicatif, et l'autre est dédié à la partie contrôle de Kubernetes, afin que les deux soient utilisés. Si l'un des deux vient à tomber en panne, l'autre prend le relais.

Un pare-feu frontal a également été mis en place afin de filtrer les ports et n'autoriser que ceux utilisés par l'infrastructure. En ce qui concerne le matériel, il s'agit d'un pare-feu récupéré par un membre de l'équipe (Paul), mais il n'a pu en récupérer qu'un seul. On supposera donc pour cette preuve de concept qu'il est possible d'en ajouter un second pour la tolérance aux pannes (en utilisant à nouveau VRRP[9], ou d'autres solutions comme CARP + pfSync[10] si la solution tourne sous pfSense par exemple). Une autre solution possible serait de mettre en place des règles iptable sur nos Raspberry pour filtrer les ports et effectuer du

NAT si besoin, mais nous n'avons pas eu le temps de mettre cela en place. L'utilisation d'un équipement séparé est également plus représentatif d'un cas réel selon nous.

Nos instances de HAProxy servent également à mitiger les attaques DDoS, en maintenant des tables qui associent les adresses IP entrantes à un taux de connexions TCP initiées. Si une IP essaye d'initier trop de connexions en un court laps de temps, ces connexions sont automatiquement rejetées.

1.3.2 Site Datacenter

Dans l'optique garder une architecture cohérente entre nos 2 sites, nous avons opté pour un fonctionnement globalement similaire sur le datacenter.

Comme illustré sur la figure 2, la répartition des VM est similaire : 3 noeuds maître pour Kubernetes, et 3 noeuds de calcul, qui jouent également le rôle de noeuds CockroachDB.

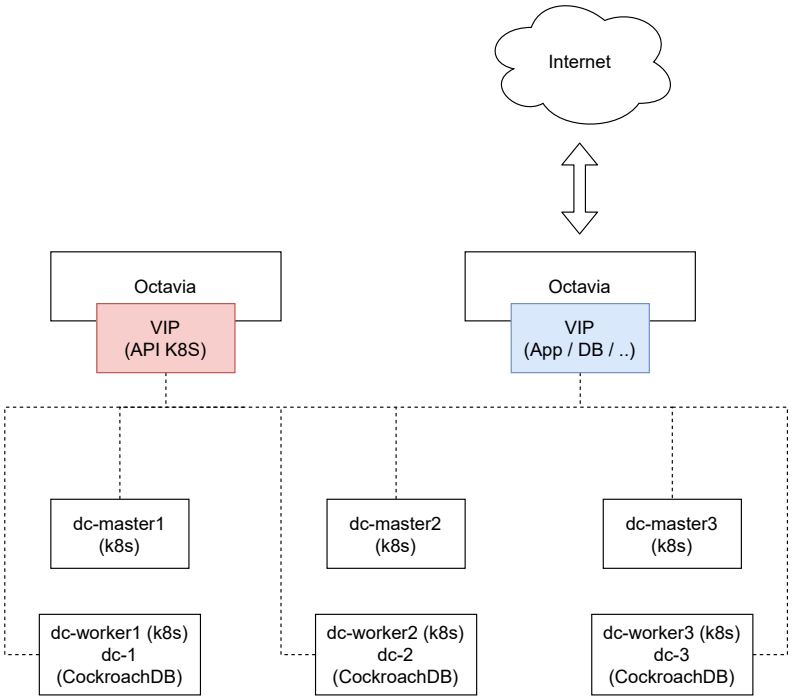

FIGURE 2 – Déploiement sur le site DC (Datacenter)

Ici, nous n'avons pas utilisé HAProxy pour les répartiteurs de charge, mais ceux proposés directement par OpenStack (Octavia).

Par ailleurs, nous avons utilisé judicieusement les groupes de sécurité d'OpenStack afin de filtrer le trafic entrant.

Enfin, il est évident que si la plateforme place nos 3 VMs de contrôle sur le même noeud physique, cela présente un risque majeur en cas de panne de l'hyperviseur. Pour tenir compte de cette possibilité, nous avons mis en place des groupes d'affinité, afin de répartir (si possible) nos VMs sur différents noeuds physiques au sein du datacenter.

1.4 Architecture logicielle

1.4.1 Kubernetes et partie applicative

Comme évoqué précédemment, notre solution est basée sur Kubernetes.

La figure 3 détaille le cheminement d'une requête d'un client vers notre application. Le point d'entrée est notre répartiteur de charge de niveau 4 (HAProxy ou Octavia), qui distribue ensuite la requête à un des noeuds de calcul Kubernetes.

La requête passe ensuite par un mécanisme d'entrée appelé "Ingress" dans Kubernetes, qui va se baser sur différents critères, dont l'URL de la requête principalement, pour diriger cette dernière vers les instances applicatives correspondantes. Les ressources ingress nécessitent l'utilisation d'un outil supplémentaire appelé ingress controller, qui permet notamment d'ajouter différentes fonctionnalités à cette étape, comme la gestion de la terminaison SSL par exemple. Deux solutions sont particulièrement populaires pour cette partie, NGINX (plus connu pour son utilisation en tant que serveur web), et Traefik[11]. Nous avons opté pour la seconde option, étant donné qu'elle est très flexible et extensible via des intergiciels (middlewares). De plus, il s'agit d'un outil d'origine française (tout comme HAProxy et Keepalived), ce qui est toujours appréciable.

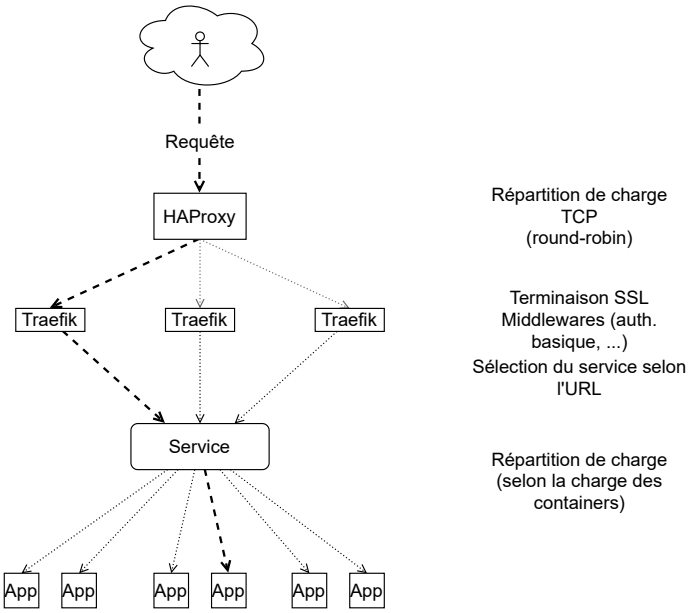

FIGURE 3 – Cheminement d'une requête

Une fois les différents intergiciels appliqués et la terminaison SSL traitée, Traefik redirige la requête vers un service Kubernetes. Les services sont en quelque sorte des répartiteurs de charge qui redirigent les requêtes vers les différentes instances (conteneurs) de l'application concernée. Ce cheminement est résumé sur la figure 4. Cette architecture nous offre donc une meilleure tolérance à la charge, de par la répartition à deux niveaux distincts.

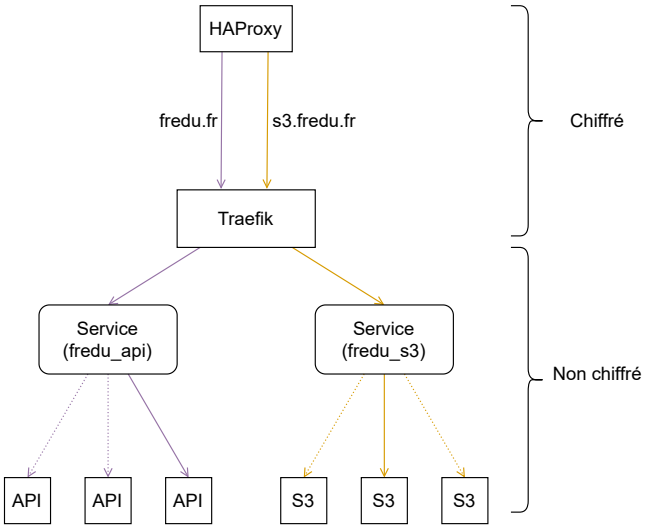

FIGURE 4 – Sélection d'une application

La gestion du nombre de répliques d'une application est dynamique et géré via l'ordonnanceur de Kubernetes. Une ressource Kubernetes appelée déploiement est en charge de la gestion des différents pods (ressource Kubernetes exécutant des conteneurs, généralement un seul par pod). Il suffit alors d'ajuster la configuration du déploiement pour adapter automatiquement le nombre de répliques au sein du cluster.

Nous avons également greffé des horizontal pod autoscaler sur certaines parties de l'application (serveur nginx pour les fichiers statiques), permettant d'ajuster dynamiquement le nombre de répliques d'une application sans intervention humaine. Ces ajustements se font selon certaines métriques, ici l'utilisation de la RAM et du CPU.

1.4.2 CockroachDB

Comme évoqué dans la section 1.2.2, notre choix pour la base de données s'est porté sur CockroachDB, dont le déploiement a été effectué sur 3 sites. Le troisième site n'est pas un site applicatif à proprement parler, les instances que nous avons loué chez le fournisseur ne servent qu'à faire tourner CockroachDB pour assurer un quorum à l'échelle globale. Dans le cadre de cette preuve de concept, ce site ne sert donc qu'à illuster le fonctionnement de CockroachDB et n'est donc pas concerné par toutes les autres considérations apportées au projet (Kubernetes, Ceph, ...).

La figure 5 illustre l'infrastructure CockroachDB de notre projet. Chaque site dispose de 3 noeuds, permettant ainsi de tolérer une panne à l'échelle locale. Les noeuds de chaque site sont reliés à deux noeuds de chacun des autres sites (par exemple, les noeuds dc sont reliés aux noeuds do1, do2, sxb1 et sxb3). Ces interconnexions permettent donc de maintenir la liaison si un des noeuds d'un des site tombe en panne (par exemple, si sxb3 tombe, la liaison avec sxb1 est toujours active). Si deux noeuds viennent à tomber sur un même site, le quorum ne serait pas respecté à l'échelle du site et il serait considéré comme défaillant de toute manière.

FIGURE 5 – Déploiement de CockroachDB

Un répartiteur de charge est placé devant les instances de chaque site afin de répartir la charge entre les différents noeuds et de s'assurer de la haute-disponibilité si un noeud est défaillant.

Les communications entre le client et les noeuds, tout comme les communications inter-noeuds, sont chiffrées à l'aide de certificats TLS. Il est impossible de se connecter à une base de données via un mot de passe, il faut obligatoirement disposer de certificats dédiés.

Pour limiter le nombre d'ouvertures de connexions vers la base de données, nous avons utilisé un pooler de connexions, afin de maintenir un certain nombre de connexions actives et réutilisables par Django. Le pooler qui est utilisé ici est pgbouncer, qui est initialement prévu pour être utilisé avec PostgreSQL, mais qui est tout-à-fait applicable ici étant donné la compatibilité étroite entre les deux SGBD.

Enfin, bien que CockroachDB soit une solution tolérante à la panne par défaut, il est toujours préférable de prévoir des sauvegardes en cas de défaillance générale. De fait, des sauvegardes récurrentes de la base de données sont programmées de façon journalière vers du stockage S3.

1.4.3 Stockage objet

Concernant le stockage objet, nous avons déployé un cluster Ceph au sein de notre infrastructure Kubernetes. Les différents composants d'un cluster Ceph, tels que les MON (Monitor Daemon) ou les OSD (Object Storage Daemon) sont donc conteneurisés, et présents sur chaque noeud de calcul. De plus, les éléments évoqués dans la section 1.4.1 sont également applicables. Le stockage objet de Ceph se base sur un moteur appelé RADOS (Reliable Autonomic Distributed Object Store), et qui est compatible avec l'API S3 d'Amazon via une passerelle appelée RADOS Gateway (RGW) et exposée en HTTP(S). Nous pouvons donc utiliser les services de Kubernetes afin de répartir la charge entre les différentes instances de la passerelle, tout en nous servant de Traefik pour gérer la terminaison SSL.

Ceph ne propose pas de solution entièrement distribuée sur plusieurs sites, mais dispose toutefois d'un mécanisme de réplication afin d'assurer une reprise après sinistre (disaster recovery), dont la mise en place sur notre infrastructure est illustré sur la figure 6. Une zone primaire est définie sur le site en C315, sur laquelle

il est possible d'effectuer des opérations de lecture et d'écriture. La zone secondaire autorise quant-à-elle uniquement la lecture. En cas de sinistre, il est toutefois actuellement nécessaire d'intervenir manuellement afin de procéder à l'élection d'une nouvelle zone primaire.

Il est à noter que la figure se concentre sur Ceph, et ne précise pas les éléments intermédiaires comme les répartiteurs de charge ou encore les ressources Kubernetes (service, Traefik).

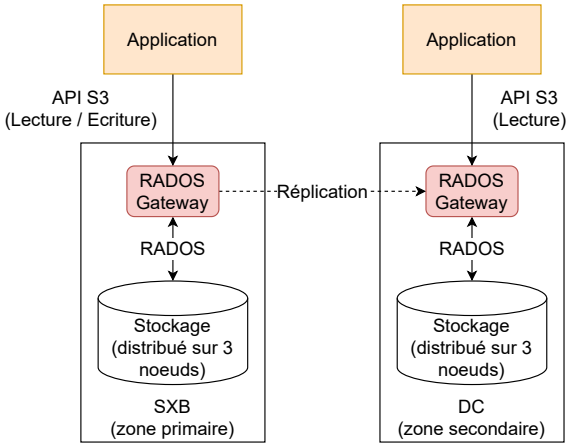

FIGURE 6 – Déploiement de Ceph / S3

1.5 Sécurisation de la solution

L'exposition de l'infrastructure a été limitée dans la mesure du possible. Seuls les ports nécessaires au bon fonctionnement de l'application sont ouverts sur les pare-feu frontaux. L'accès à l'application et au stockage objet se fait uniquement via HTTPS, forçant ainsi le chiffrement des données. Comme les accès à la base de données peuvent être soit locaux, soit distants, les requêtes vers la base de données sont également chiffrées. De même, les communications inter-noeuds sont vérifiées à l'aide de certificats, empêchant ainsi des nuds étrangers de joindre le cluster.

Notre service étant sous la forme d'une plateforme web, l'un des risques les plus évidents est le déni de service. Pour limiter ce dernier, nous avons mis en place plusieurs mécanismes de sécurité.

En C315, le point d'entrée étant notre répartiteur de charge HAProxy, nous avons configuré ce dernier en conséquence via les "stick tables" de l'outil. Cette directive permet de maintenir une table avec les différentes adresses IP accédant au proxy, pour y appliquer différents filtres. Ici, nous avons choisi de limiter le nombre de connexions TCP dans un intervalle de temps donné pour une adresse IP. Si trop de connexions sont initiées dans un court laps de temps, la plupart seront rejetées.

Concernant notre déploiement sur le datacenter, les répartiteurs de charge d'OpenStack ne disposent pas encore de mécanisme de mitigation des DDoS. De plus, ils fonctionnent en mode full-proxy, l'IP source est réécrite, il n'est donc pas possible de se fier à cette dernière pour les blocages à d'autres niveaux. Nous avons tout de même essayé de limiter les risques, en introduisant une limite de requêtes brute via Traefik, cette fois-ci au niveau 7 en renvoyant un code HTTP 429 quand la limite est atteinte. Il s'agit cependant d'une solution nettement moins flexible, mais seule alternative possible compte-tenu de notre architecture. Une solution éventuelle aurait été de déployer nos propres répartiteurs de charge via des VM et HAProxy, mais le quota alloué pour les vCPU et la RAM est trop juste.

L'API de Kubernetes n'est pas exposée directement non plus, il est nécessaire de passer par un bastion afin d'y accéder, et ce pour nos deux sites.

FIGURE 7 – Dashboard Grafana avec des métriques Prometheus

1.6 Support double-pile

Le datacenter de l'Université de Strasbourg ne nous permettant pas de disposer d'adresses IPv6, il n'est donc pas concerné par cette problématique. Notre site placé en C315 utilisant HAProxy en tant que point d'entrée principal vers l'infrastructure (afin de répartir la charge), tout le trafic extérieur passe donc par un proxy inverse. Notre application est donc accessible en IPv6 depuis l'extérieur, HAProxy écoutant également sur une adresse IPv6. Les mécanismes de mitigation de DDoS sont évidemment présents en IPv6 également, avec des tables dédiées.

1.7 Supervision de l'infrastructure

Différents outils de monitoring ont été déployés sur l'infrastructure. Via Kubernetes, nous avons déployé des instances de Prometheus[12] ainsi que des exporteurs afin d'obtenir des métriques relatives aux différents noeuds. La visualisation des données est assurée via l'outil Grafana[13]. Cela nous permet donc de visualiser différentes données telles que l'utilisation globale des noeuds (utilisation du CPU, utilisation de la RAM, comme le montre la figure 7), mais également des données plus ciblées, à l'échelle des différents conteneurs.

Le cluster Ceph dispose de son propre dashboard, permettant de visualiser l'état des noeuds et la capacité de stockage, mais également de gérer les différents utilisateurs et buckets S3.

Il en est de même pour CockroachDB. Le dashboard de ce dernier nous permet non seulement de visualiser des informations relatives au stockage et à l'état des noeuds, mais également d'obtenir des informations relatives aux requêtes effectuées, comme le temps de réponse moyen. Cette fonctionnalité nous a notamment permis de résoudre certains problèmes de latence, les statistiques relatives aux requêtes exécutées ayant révélé que certaines requêtes étaient exécutées beaucoup trop souvent à cause d'une mauvaise utilisation de l'ORM de Django. La figure 8 illustre ce cas de figure, où lors de nos tests de charge, nous avons remarqué qu'une requête était effectuée beaucoup plus de fois que les autres pour un test en particulier.

1.8 Performances de l'application

Notre application étant répliquée sur plusieurs noeuds ainsi que sur plusieurs sites, la répartition de charge nous permet techniquement de traiter une charge importante.

STATEMENTS	TXN TYPE	RETRIES	EXECUTION COUNT	ROWS AFFECTED
SELECT FROM users_teacher	Implicit	0	60k	0
SELECT FROM courses_course	Implicit	0	11k	1
SELECT FROM courses_course	Implicit	0	11k	6
SELECT FROM users_teacher	Implicit	0	11k	1
SELECT FROM users_user	Implicit	0	11k	1
SELECT FROM users_user	Implicit	0	3k	1
SELECT FROM django_session	Implicit	0	3k	1
SELECT FROM users_teacher	Implicit	0	3k	1
INSERT INTO courses_course	Implicit	0	3k	1

FIGURE 8 – Analyse des requêtes SQL

Selon l'infrastructure réseau sous-jacente, la séparation en plusieurs sites pourrait également avoir des effets bénéfiques en terme de latence, en permettant à un utilisateur de joindre le site le plus proche de sa position, soit manuellement en utilisant un domaine spécifique à un site (sxb.fredu.fr par exemple), soit idéalement de manière transparente via un domaine générique (fredu.fr). Cette dernière option nécessite toutefois de pouvoir mettre en place de l'anycast au niveau des deux sites.

Afin de vérifier le bon fonctionnement de notre infrastructure, nous avons effectué des tests de charge en lecture grâce à l'outil k6[14]. Les tests ont été effectués depuis une machine distante (le RTT entre la machine et les deux sites étant d'environ 16ms), pour 10, 20, 50 et 100 "utilisateurs virtuels", qui sont l'équivalent de threads pour k6. Le test qui a été effectué consiste à récupérer un cours au hasard (ses données, ainsi que les informations relatives aux auteurs) parmi 10 en interrogeant l'API. Chaque série de tests dure 1 minute.

Un récapitulatif du nombre de requêtes effectuées par secondes selon les cas de test est disponible sur la figure 9.

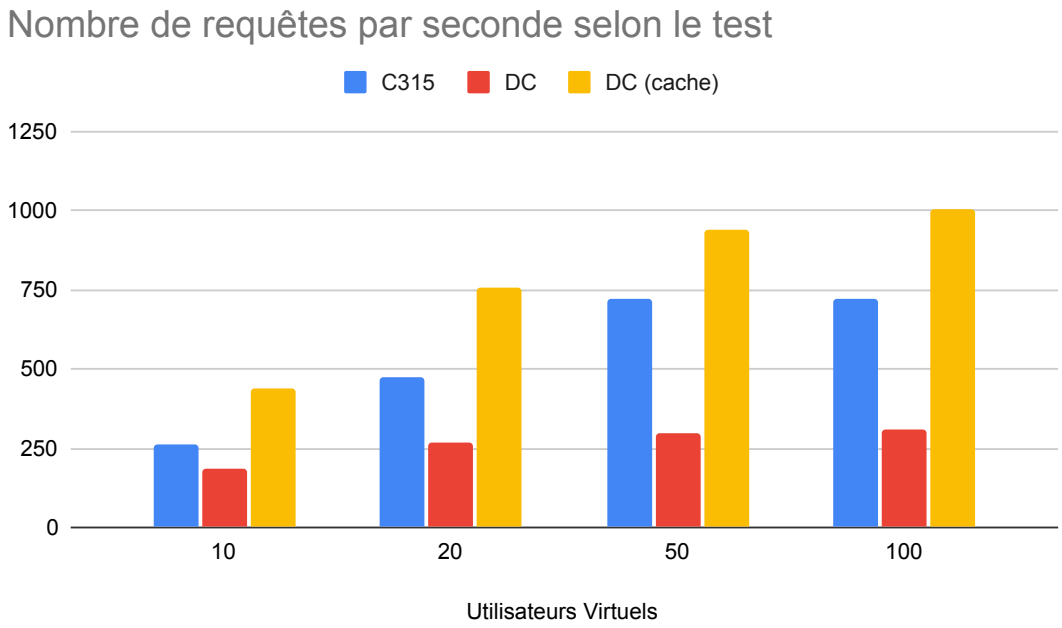

FIGURE 9 – Requêtes effectuées par seconde selon les tests

Nous avons commencé par tester le site en C315, dont les résultats sont visibles sur la figure 10. Nous avons pu constater des temps de réponse plutôt bons, de l'ordre de 40 ms en moyenne pour 10 à 20 threads. Pour 50 threads, le temps de réponse est également bon, avec 95% des requêtes qui sont servies aux alentours des 100ms. Les résultats fournis par k6 ne prenant pas en compte le RTT, si l'on soustrait les 16ms de latence, on obtient des résultats convaincants, le test à 50 threads passant même sous les 100ms symboliques (sensation d'instantanéité) pour 95% des requêtes. Pour 100 threads, quelques requêtes témoignent d'une certaine latence, mais la moyenne aux alentours des 120ms (RTT soustrait) semble acceptable pour environ 700 requêtes à la seconde.

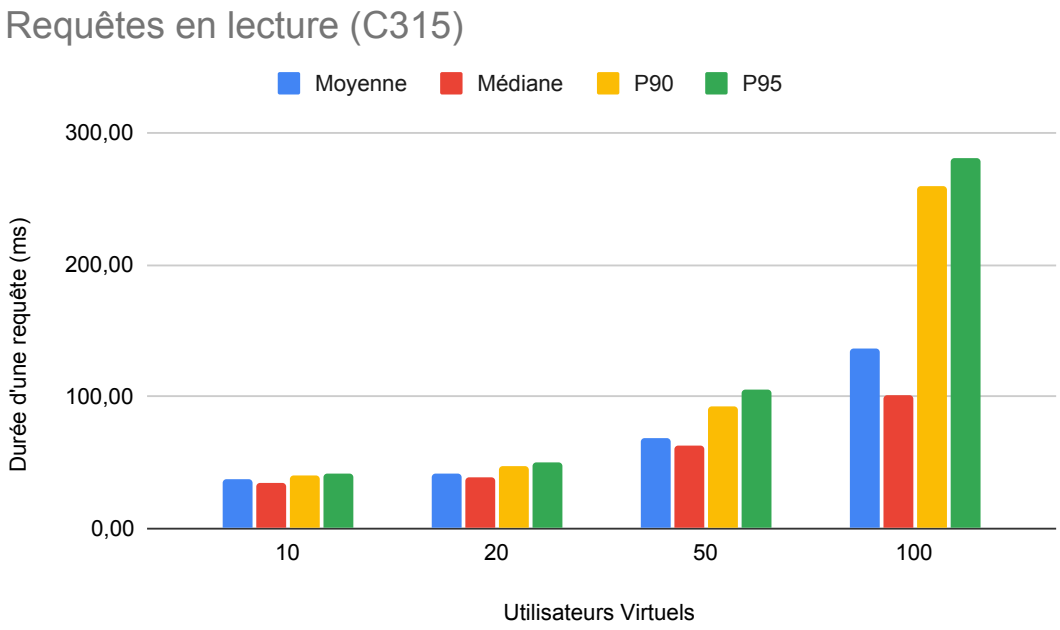

FIGURE 10 – Requêtes vers la C315

Nous avons également effectué les tests sur le datacenter, en suivant le même processus. Les résultats obtenus sont visibles sur la figure 11. On constate que les performances sont moins bonnes, mais toujours correctes pour 10 et 20 threads, et acceptables pour 50 threads en moyenne (le test à 100 threads n'est pas montré sur le graphique par soucis d'échelle, mais comme attendu, les performances sont nettement inférieures à celles de la C315 avec un temps moyen de réponse aux alentours de 300ms).

Nous pouvons émettre plusieurs hypothèses sur ces différences. D'une part, l'architecture de CockroachDB fonctionne grâce à des éléments appelés "leaseholders", qui sont chargé de coordonner les lectures et les écritures au sein de la base de données (un par table). Les lectures sont donc tributaires de la latence entre le noeud interrogé par l'application (local) et le leaseholder de la table concernée (pas forcément local). Cette hypothèse est toutefois écartée dans notre cas, étant donné que notre configuration place les leaseholders sur les noeuds en C315 si possible, et que le backbone entre les deux sites est partagé, la latence est donc minime.

La deuxième hypothèse, qui cette fois est probablement correcte, concerne la différence de matériel entre les deux sites. En effet, les VMs de calcul du datacenter disposent de ressources plus limitées que les noeuds présents en C315, que ce soit en terme de RAM ou de CPU.

Les tests de charge que nous avons pu effectuer sur notre infrastructure témoignent d'une certaine latence lorsque le nombre de requêtes est élévé, dont certaines causes sont les suivantes peuvent être les suivantes :

— nous n'avons pas eu le temps d'optimiser la base de données, c'est-à-dire l'indexer correctement

— certaines requêtes de l'application via l'ORM de Django sont mal configurées (plus de requêtes vers la base de données que nécessaire, ce qui multiplié par le nombre de requêtes en parallèle ajoute un certain coût)
— il s'agit d'une preuve de concept dont tous les services tournent sur un seul noeud, il faudrait idéalement des ressources dédiées, a minima pour séparer la base de données de l'application web

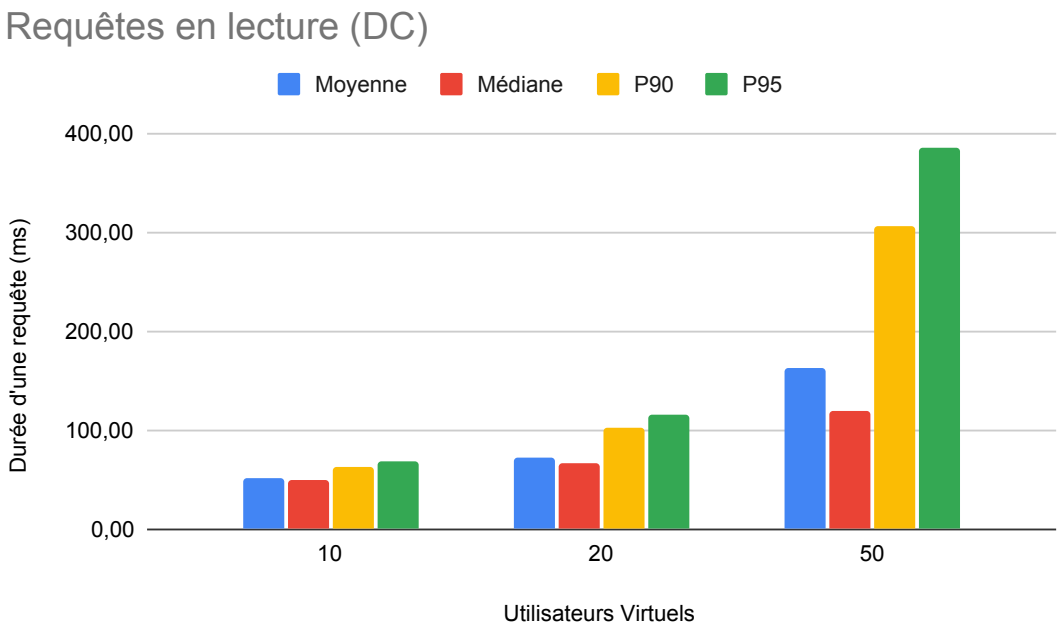

FIGURE 11 – Requêtes vers le datacenter

Toutefois, afin d'alléger la charge de la base de données, nous avons considéré certaines requêtes comme peu sensibles et utilisé une infrastructure de cache (via l'outil memcached) pour ces dernières. Par exemple, nous pouvons considérer qu'afficher la liste des cours disponibles ne nécessite pas d'être tout le temps à jour, quelques minutes de décalage sont tolérables (surtout compte tenu du fait qu'il s'agit d'une requête effectuée régulièrement, et qui croise plusieurs tables). Une mise en cache avec une durée de 2 ou 3 minutes nous a donc semblé pertinente à ce niveau.

L'utilisation du cache nous permet d'obtenir de très bonnes performances pour les requêtes mises en cache, ainsi que pour les parties statiques du site (fichiers JavaScript). Nous avons effectué quelques tests vers le datacenter, en effectuant le même test que précedemment, mais avec une mise en cache des requêtes activée. Les résultats sont visibles sur la figure 12. On constate un temps de réponse assez bas, ce qui est effectivement attendu grâce au cache. Jusqu'à 50 threads, 90% des requêtes sont servies en moins de 100ms (voire 95% si le RTT est soustrait), ce qui est la valeur cible à atteindre pour avoir une sensation d'instantanéité. Pour 100 threads, certaines requêtes sont nettement plus longues à traiter, mais la durée moyenne reste plus que correcte, surtout qu'ici nous atteignons la barre des 1000 requêtes par seconde en moyenne. Cette augmentation peut être due à la taille de l'infrastructure, ici chaque noeud faisant tourner tous les services, ou bien aux ressources allouées aux répartiteurs de charge (nous n'avons pas d'informations dessus). Une autre explication serait le fait que Django n'est pas multi-threadé par défaut, l'utilisation des fonctionnalités asynchrones de Python et Django étant peut être une piste à explorer également.

Il est toutefois important de noter qu'en pratique, les requêtes utilisant la base de données sont tributaires de la latence entre le noeud local et le leaseholder de la table interrogée. Ainsi, si les leaseholders n'étaient pas placés localement, les performances s'en trouveraient fortement impactées. CockroachDB dispose toutefois d'un mécanisme de géo-partitionnement des données, permettant via divers mécanismes d'améliorer les performances locales. Une amélioration du système actuel est donc possible, notamment via un mécanisme

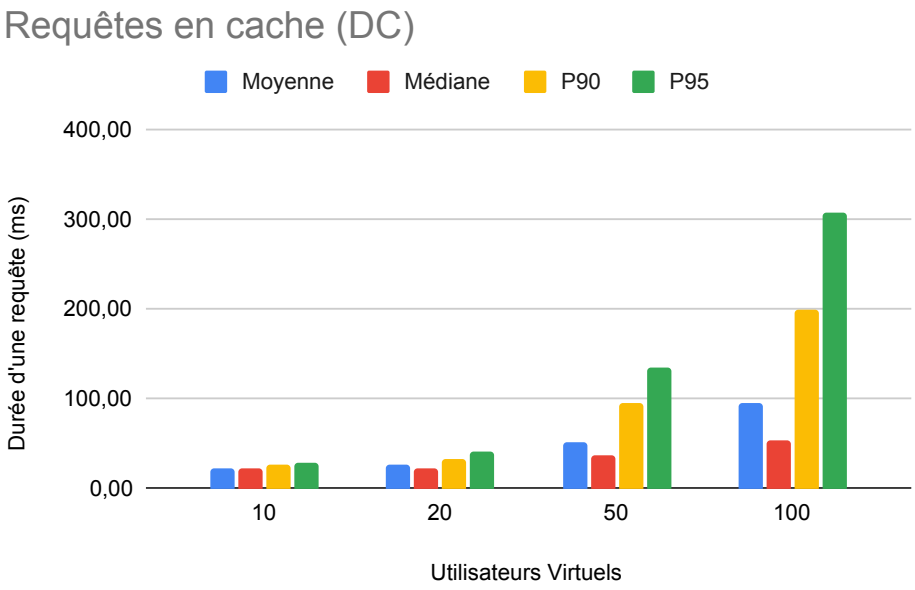

FIGURE 12 – Requêtes en cache vers le datacenter

d'index dupliqués, qui permet, comme son nom le suggère, de dupliquer les données indexées sur chaque région. Naturellement, toutes les méthodes de géo-partitionnement ont une contrepartie, qui est généralement un temps nécessaire à l'écriture qui est plus important (comme pour l'indexation en général), mais étant donné la nature de notre application, ce serait totalement acceptable (le nombre de lectures pour un cours sera bien plus élevé que le nombre de fois où il sera modifié, par exemple).

Notre infrastructure déployée a l'avantage d'avoir des latences minimes entre nos deux sites, et notre troisième site n'est pas réellement utilisé, dans le sens où notre placement des leaseholders est basé sur nos deux sites applicatifs en priorité, et qu'il n'héberge que les instances de CockroachDB. On pourrait dire qu'il joue simplement un rôle similaire à celui d'un arbitre. Notre projet est cependant pensé pour être déployé à une échelle nationale, ce qui implique forcément des latences plus élevées entre les différents sites, et dans ce cas de figure, il sera nécessaire de mettre en place ces mécanismes de géo-partitionnement afin de garantir un fonctionnement optimal de l'application.

Enfin, si la plate-forme reçoit beaucoup de requêtes, c'est qu'il y a beaucoup d'utilisateurs, et par conséquent beaucoup de données. Comme évoqué précédemment, notre approche a été de viser la scalabilité horizontale, et CockroachDB correspond bien à nos besoins ici de par son architecture interne grâce à un mécanisme de fragmentation automatique des données. Ces dernières sont donc naturellement réparties sur plusieurs noeuds.

1.9 Démarche Infrastructure as Code

Le déploiement de l'architecture a été effectué dans une démarche Infrastructure as Code grâce à différents outils, afin de versionner la plus grande partie de l'infrastructure.

Le déploiement des clusters Kubernetes a été assuré via kubespray, qui est un ensemble de playbooks Ansible[15]. D'autres playbooks et rôles Ansible ont été écrits pour la configuration des répartiteurs de charge HAProxy (+ Keepalived), ainsi que pour l'installation et la configuration de CockroachDB.

Kubernetes fonctionnant par le biais de ressources déclaratives écrites sous forme de fichiers YAML, la gestion des conteneurs applicatifs est naturellement orientée Infrastructure as Code également.

Nom	Total	Infra.	Back.	Front.	Gestion	. Autre
Alexandre	51	0	0	0	51	0
Hussein	66.5	0	64.5	0	0	2
Oussema	84	0	0	78	0	6
Paul	144	108	16	9	7	4
Robin	52	0	38	0	0	14

TABLE 1 – Estimation du temps de travail (en heures)

Enfin, l'approvisionnement des machines virtuelles dans le datacenter a été effectué grâce à Terraform[16]. Aucune action manuelle n'a été effectuée depuis la plateforme web d'OpenStack (si ce n'est pour récupérer les identifiants, naturellement). La configuration Terraform présente sur notre Git est complète, et gère les différentes VMs, les volumes, mais également toute la partie réseau (adressage, IP flottantes, groupes de sécurité), les répartiteurs de charge ainsi que les groupes d'affinité.

L'utilisation de ces outils nous a été particulièrement bénéfique car il y a eu une migration de la plateforme OpenStack courant novembre, et le temps passé au à la rédaction des ressources Ansible / Terraform nous a permis de re-déployer l'infrastructure sur le datacenter dans des très brefs délais (et surtout, à l'identique).

2 Gestion de projet

2.1 Organisation du projet

L'équipe du projet à été organisée en 5 branches de sorte à englober l'ensemble des aspects du projet :

- Le déploiement de l'infrastructure a été assuré par Paul
- La mise en place des outils de partage de travail par Robin
- Le backend de la solution par Hussein
- Le frontend de la solution par Oussema
- Les tâches de gestion du projet par Alexandre

Un Git décomposé en plusieurs branches à été mis en place pour avoir une vue des modifications apportés par chacun.

Une fiche partagée intitulée "Enregistrement des temps de travail"[17] est à disposition de l'équipe pour avoir une représentation précise du temps de travail fourni à la réalisation du projet. Une estimation du temps de travail de chaque membre est disponible sur la table 1.

Un diagramme de Gantt détaillé à été mis en place pour suivre la progression de l'équipe.

Le client et le chef de projet ont effectué des entretiens bi-mensuels lors desquelles plusieurs aspects du projet ont été discutés notamment :

- La gestion de l'équipe
- La progression du projet
- Les problèmes techniques et humains rencontrés
- Les souhaits du client et les rectifications désirés

Suite aux entretiens clients, des entretiens équipes ont été tenus de sorte à diffuser les retours du client parmi tous les membres de l'équipe et pour donner la possibilité de discuter des problèmes rencontrés et des questions à poser. Ces entretiens ont très fortement participé à la communication au sein de l'équipe et à la compréhension du besoin client.

Le chef de projet a aussi documenté de façon systématique chaque entretien pour avoir une trace des sujets discutés et des conclusions apportées.

Un chat discord a été mis en place pour permettre la communication informelle entre les membres de

l'équipe.

2.2 Méthode de travail

La méthode de travail adoptée est le travail individuel avec un retour dans le chat discord à la fin d'une fonctionnalité implémentée.

Courant novembre des séances de travail par binôme ont été instaurées pour permettre un partage de savoir et un travail effectif sur les parties du projet nécessitant de relier deux composantes de la solution développées séparément (back-end et front-end notamment).

La situation sanitaire particulière de cette année a rendu impossible le travail par groupe en présentiel. L'équipe a donc opté pour des séances en vocal avec des microphones allumés pour avoir un travail plus convivial. Ces séances ont eu lieu depuis mi-décembre.

2.3 Gestion du temps de travail

Un planning de Gantt a été proposé au début du projet et suivi tout au long de sa réalisation. Il a été mis à jour lors des retards et affiné tout au long de l'avancée.

La figure 13 représente le diagramme de Gantt de l'équipe au moment de la rédaction du mémoire. On peut distinguer 5 parties représentés par des couleurs différentes :

- Cyan : 14/09 - 03/10. Rédaction du cahier des charges ainsi que de la réponse par l'ensemble de l'équipe.
- Rouge : 04/10 - 11/01. La partie déploiement de l'infrastructure.
- Bleu : 04/10 - 11/01. La partie backend.
- Violet : 04/10 - 05/01. La partie frontend.
- Jaune : 15/11 - 10/01. La partie sur les outils de multi-authoring.

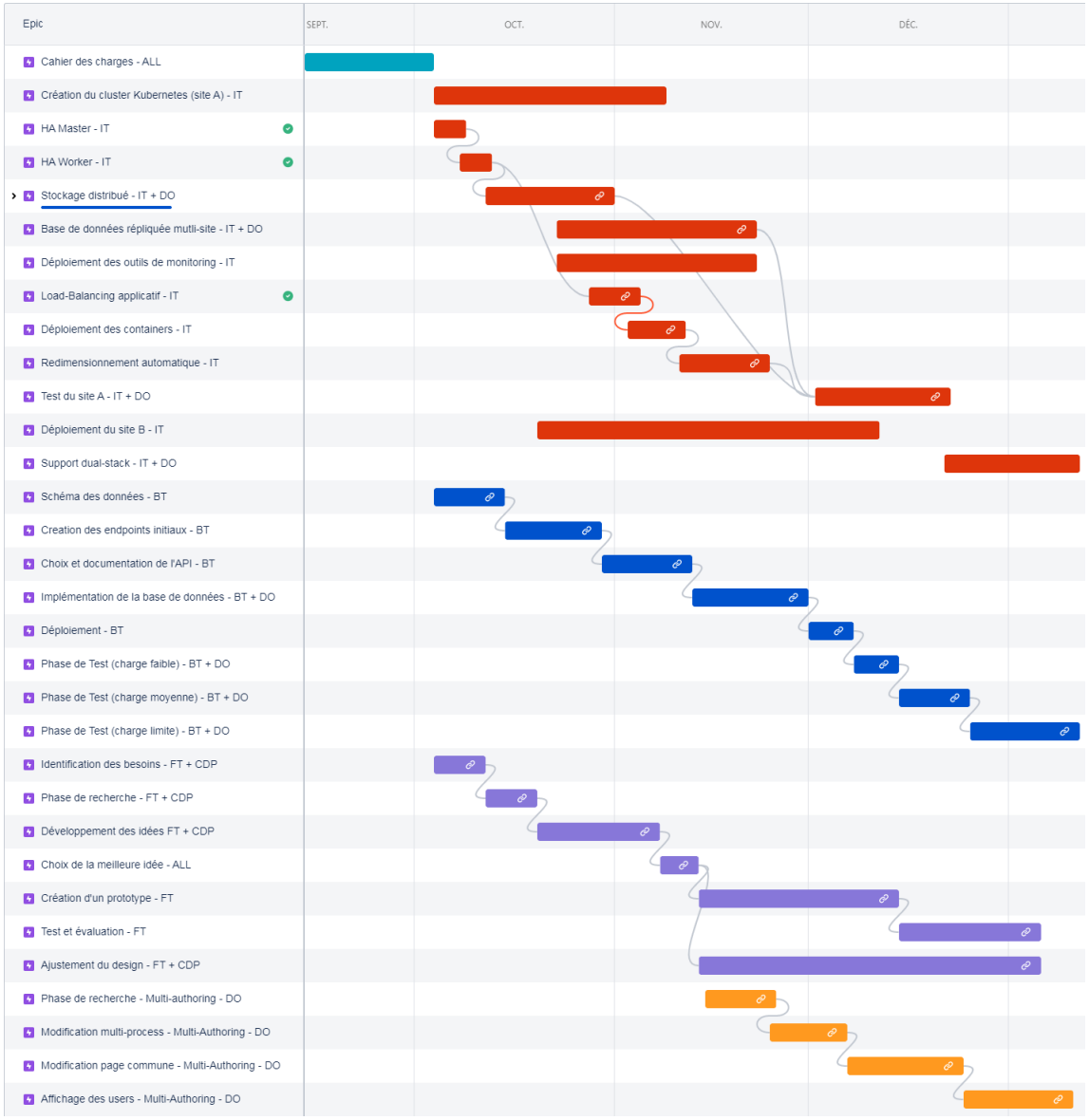

FIGURE 13 – Diagramme de GANTT

3 Résultat final

3.1 Ce qui a été réalisé

En l'état actuel, nous avons un systèmes d'utilisateur complet. Celui-ci permet la création d'un utilisateur de rôle "Enseignant" ou "Étudiant". Le premier rôle permet à l'utilisateur de créer un cours et le rendre disponible sur notre plate-forme. Tout deux peuvent accéder aux cours d'autrui si ils y sont invités ou que le cours est publique. On peut voir la page d'un utilisateur inscrit à des cours sur la figure 15 ou la liste pour s'inscrire a des cours sur la figure 14.

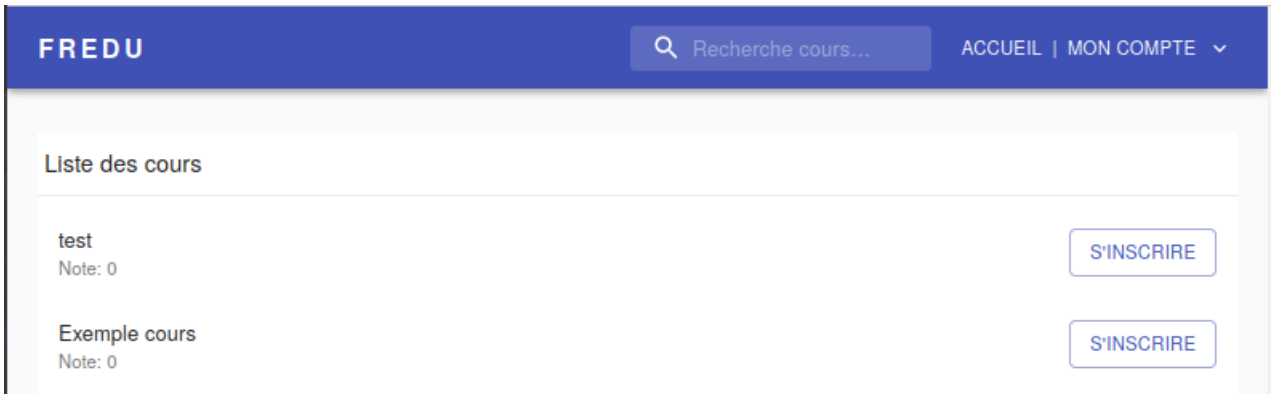

FIGURE 14 – Liste des cours pour s'inscrire.

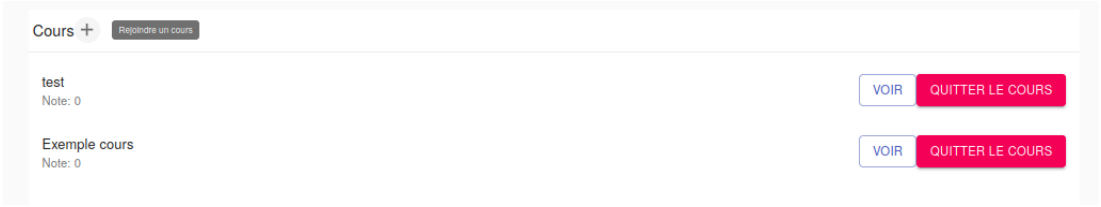

FIGURE 15 – Exemple de cours d'un utilisateur.

Un utilisateur à la possibilité de changer son mot de passe dans la catégorie "Mon compte".

Dans un cours, il est possible d'y avoir des documents de multiples natures. Ceux-ci seront téléchargeables pour la lecture en local. En plus de ces documents, on peut y lire le texte du cours qui peut être fourni sous forme de Markdown ou de simple texte. On peut voir le résultat d'un de ses cours sur la figure 16. Nous avons également mis en place une syntaxe spécifique, lue lors de la traduction du Markdown en HTML, afin de référencer les fichiers qui ont été mis en ligne (au lieu d'avoir à spécifier manuellement l'URL S3 de la ressource).

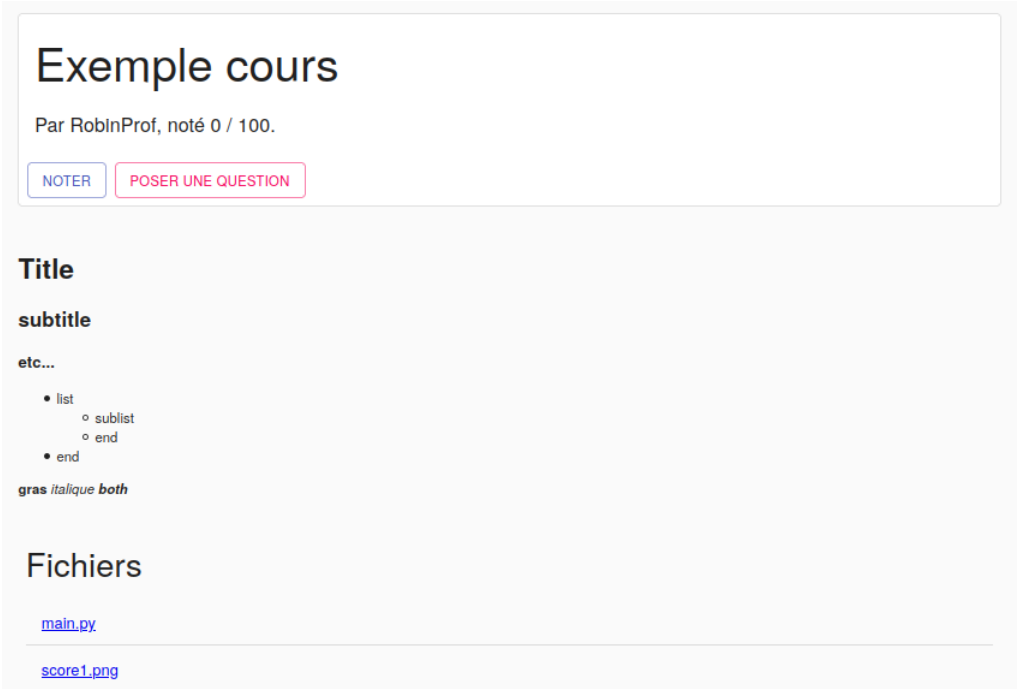

FIGURE 16 – Exemple de cours contenant des fichiers.

Afin de permettre l'interactivité quant aux cours disponibles sur la plate-forme, un système de questions a été mis en place. Celles-ci sont accessibles à la fin de chaque cours. Par exemple on peut voir une question posée sur un cours en figure 17. Sur celui-ci il est possible d'échanger en posant des questions et en y recevant les réponses dans des discussions, comme on peut le voir sur la figure 18, ces dernières étant évidemment spécifiques à chaque cours.

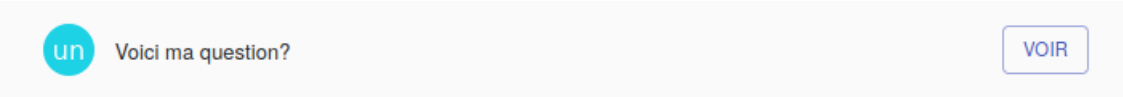

FIGURE 17 – Vue des questions dans le cours

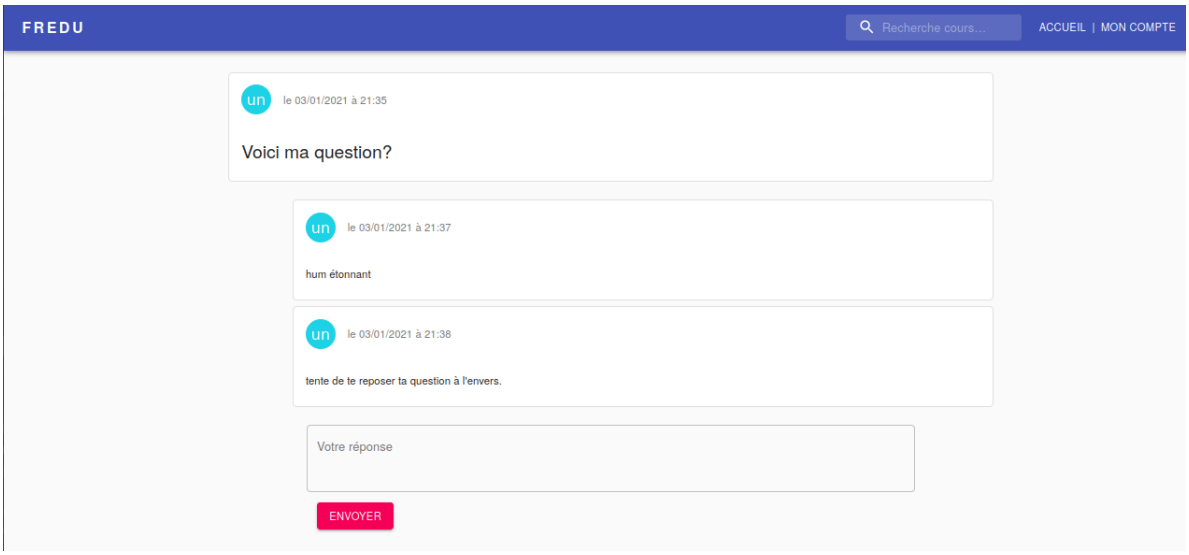

FIGURE 18 – Exemple de forum lié à un cours

3.2 Ce qui n'a pu être réalisé

Au cours du projet, il a été prévu de mettre en place un système d'édition à auteurs multiples sur les cours. Cette fonctionnalité a été abandonné car il était impossible de réunir les gens nécessaires pour finir cette fonctionnalité sans mettre à mal la finition des fonctionnalités importantes de notre outil.

Il est encore possible d'optimiser les requêtes avec un indexage plus poussé ou encore une meilleure répartition des données. De cette façon l'utilisateur aura un accès plus rapide à chacune des informations qui lui sont nécessaires.

Enfin, une partie des fonctionnalités de l'application initialement présentes dans le cahier des charges n'a pas été implémentée.

3.3 Les difficultés rencontrées

Le projet a pris un grand retard. Ce retard a en partie été causé par la situation sanitaire : la motivation de certains membres de l'équipe s'en est vu fortement impactée par le travail à distance. Les différentes fonctionnalités à implémenter ont donc accumulé un retard important (lié au manque de motivation mais également au manque de travail régulier de la part d'une partie de l'équipe), nous avons donc été obligé de revoir nos prévisions à la baisse. Le confinement a également posé des petits soucis quant aux délais de maintenance de l'infrastructure quand des contraintes sont apparues, empêchant le travail sur l'un des sites pendant quelques jours.

4 Bilan

Pour conclure, à travers ce projet, nous avons travaillé sur un projet avec une vraie dimension professionnelle. La rédaction d'un cahier des charges, ainsi que de sa réponse ont été des expériences nouvelles, tout comme l'adaptation d'un projet suite à diverses réunions avec un "client". Ce projet nous a également permis de nous rendre compte des difficultés organisationnelles et humaines qui peuvent survenir durant toute la période sur laquelle il s'étend. Nous avons cependant pu travailler avec des technologies de pointe, et découvrir de nouvelles solutions pour répondre aux problèmes qui se sont posés. Il est cependant dommage

qu'une partie des attentes liées au projet n'aient pu être réalisées, bien que les fonctionnalités principales aient été implémentées.

5 Références

[1] "The Web framework for perfectionists with deadlines | Django." [Online]. Available : https://www.djangoproject.com/

[2] "React Une bibliothèque JavaScript pour créer des interfaces utilisateurs." [Online]. Available : https://fr.reactjs.org/

[3] "Solution professionnelle dorchestration de conteneurs." [Online]. Available : https://kubernetes.io/fr/

[4] "Rook." [Online]. Available : https://rook.io/

[5] "Cockroach Labs, the company building CockroachDB." [Online]. Available : https://www.cockroachlabs.com/

[6] D. Ongaro and J. Ousterhout, "In Search of an Understandable Consensus Algorithm," p. 18.

[7] "HAProxy - The Reliable, High Performance TCP/HTTP Load Balancer." [Online]. Available : http://www.haproxy.org/

[8] "Keepalived for Linux." [Online]. Available : https://www.keepalived.org/

[9] R. Hinden <bob.hinden@nokia.com>, "Virtual Router Redundancy Protocol (VRRP)." [Online]. Available : https://tools.ietf.org/html/rfc3768

[10] "OpenBSD PF : Firewall Redundancy (CARP and pfsync)." [Online]. Available : https://www.openbsd.org/faq/pf/carp.html

[11] "Traefik Labs : Makes Networking Boring." [Online]. Available : https://traefik.io/

[12] "Prometheus - Monitoring system & time series database." [Online]. Available : https://prometheus.io/

[13] "Grafana : The open observability platform." [Online]. Available : /

[14] "Open-source load testing tool for developers." [Online]. Available : https://k6.io/open-source/

[15] A. Hat, Red, "Ansible is Simple IT Automation." [Online]. Available : https://www.ansible.com

[16] "Terraform by HashiCorp." [Online]. Available : https://www.terraform.io/

[17] "Enregistrement du temps de travail." [Online]. Available : https://docs.google.com/spreadsheets/d/1C6LFYkktrTGt_CYZATDoE9PKJSKFLZPLGe_QpPCzFjk/edit?usp=sharing