

**ELEVE**

Nom : Dewas

Prénom : Faustin

Filière : SE

**SUJET** : Stage Technique M1

**ENTREPRISE**

Nom : Aneo

Adresse : 122 avenue du général Leclerc 92100 Boulogne-Billancourt

**DATE DU STAGE**

Date de remise du rapport aux membres du Jury :

du : **02/11/2022** au : **21/03/2023**

durée effective en semaines : **20 semaines**

**SOUTENANCE**

Date :

Heure :

Composition du Jury :

- Président (responsable EFREI ) :
- Responsable du stage (Entreprise) :
- Invité(e) :

**PUBLICATION DU RAPPORT DE STAGE**

Le Responsable du stage : **Nicolas GRUEL** autorise le stagiaire à publier le rapport de stage sur l'Intranet de l'Ecole.

Signature



**Mots clés**

Faustin Dewas  
Groupe M1 SE1



# Stage et Technique Humaine

Rapport de stage



## Remerciements

Avant de commencer ce rapport, j'aimerais remercier l'ensemble de l'équipe ArmoniK, qui m'a accompagné pendant tout ce stage. Merci Estéban, Nicolas, Jérôme, Florian, de m'avoir aidé à développer mes connaissances pendant ce stage, et d'avoir répondu à mes interrogations.

Merci aussi à toutes les autres collaborateurs d'Aneo avec qui j'ai eu le plaisir de discuter pendant ces 20 semaines.

## I- Une toile vide

Nous pouvons, en temps qu'êtres humains, être rapportés à un tableau. Nous sommes une toile jamais finalisée, constamment en évolution, et chaque instant de notre vie est une occasion pour ajouter de nouveaux pigments de couleur sur notre toile. Pendant 5 mois, j'ai eu l'occasion d'ajouter celles d'Aneo à la mienne, la faisant évoluer d'une toute nouvelle manière. C'était la première fois que j'utilisais mes compétences en informatique pour travailler sur un projet industriel. J'ai pu travailler en mode projet, sur des technologies familières et plus ou moins étrangères.

Ce stage sur le projet ArmoniK m'a permis de découvrir de nombreuses choses, d'ajouter un panel de couleur très large à ma toile. Que ça soit à propos de la gestion de projet, du fonctionnement d'une équipe ou des différentes technologies utilisées, j'ai eu l'occasion de découvrir un véritable dégradé de compétences et eu l'occasion d'en assimiler la plupart.

Mais je ne pouvais pas me contenter d'apprendre pendant ce stage, il était nécessaire que j'ajoute ma pierre à l'édifice dans ce projet professionnel. C'est pourquoi j'ai mis un point d'honneur à mettre le plus d'efforts possible pour fournir un travail qualitatif, en utilisant des principes de développement et de travail vus à l'Efrei, mais aussi en écoutant les conseils de mes collègues.

En résumé, j'ai pu, pendant 20 semaines, rajouter des couleurs à ma palette. Ce rapport a pour but d'analyser ces-dites couleurs, de retracer mon évolution sur la compréhension et l'utilisation que j'avais d'elles.

## Sommaire

Remerciements.....	3
I - Une toile vide .....	4
II - Aneo, une véritable palette de couleurs .....	6
1. Présentation & quelques chiffres .....	6
2. Cercles et couleurs .....	7
3. Événements .....	7
4. Cadre de vie et travail.....	8
II - ArmoniK.....	10
1. Présentation du projet .....	10
2. Ma mission .....	10
3. Équipe et management .....	11
4. Intégration de Git .....	15
5. Déploiement d'ArmoniK .....	25
A. Docker, containers, images .....	25
B. Kubernetes .....	26
C. Terraform .....	27
D. Fonctionnement commun de K8s avec Terraform .....	29
6. Des outils de taille .....	29
7. Implication dans le projet.....	33
A. Seq et Grafana.....	33
B. Filtres.....	33
C. Nouvelles pages ! .....	35
D. Gestions de différents bugs et petites fonctionnalités .....	35
E. Reviews .....	35
F. Actions GitHub.....	36
G. Nginx (Ingress).....	36
H. Terraform (Double service) .....	37
8. Difficultés rencontrées .....	38
III - ArmoniK de couleur taillé dans le bois.....	39
Anecdotes.....	43
Annexe.....	44

## II- Aneo, une véritable palette de couleurs

### 1. Présentation & quelques chiffres

Aneo est un cabinet de conseil spécialisé dans la transition numérique et organisationnelle des entreprises. Fondée en 2002 par Pierre Sinodinos, l'entreprise a subi plusieurs restructurations et changements de stratégie au fil du temps, faisant d'Aneo une entreprise absolument unique. Aujourd'hui, Aneo est composée de XXX personnes, et son chiffre d'affaires de 2022 est de 22 177 600€.

Aneo a une évolution très particulière pour une entreprise. De sa création à 2009, l'entreprise était considérée comme une ESN, une Entreprise de Service du Numérique, spécialisée dans les nouvelles technologies et l'informatique. Après cette période, Aneo commença à faire évoluer ses offres, mais en éprouvant quelques difficultés organisationnelles. Finalement, Aneo est devenue une entreprise se définissant comme « hybride ». C'est-à-dire qu'en plus de fournir des services numériques, elle est maintenant capable de fournir des conseils en Informatique, en gestion de projet et en management.

Evidemment, Aneo possède des concurrents dans les divers secteurs sur lesquels elle travaille. Au niveau information et technologie, on retrouve notamment Arolla, Cellenza, Finaxys, Atos, orange business services. La concurrence sur l'organisation d'entreprises se retrouve davantage chez wemanim, bartle ou juliet sterwen. Enfin, nous pouvons retrouver parmi les principaux leaders des marchés dans lesquels Aneo se concentre capgemini, aubay, wavestone, devoteam ou encore sopra.

Un des points particuliers d'Aneo, qui la différencie de beaucoup d'autres entreprises, c'est sa gestion des collaborateurs. En effet, l'entreprise a mis en place une hiérarchie plate. C'est-à-dire que la chaîne de commandement est réduite au minimum, de manière à simplifier le plus possible les processus d'administration, en limitant le nombre d'intermédiaires. La plupart du temps, les interactions sont faites directement entre personnes concernées. Cela permet à tous les collaborateurs de ne pas se sentir délaissés et remplaçables, et de se sentir acceptés plus facilement dans l'entreprise, ce qui facilite énormément les relations et les échanges.

## 2. Cercles et couleurs

Aneo continue de recruter toujours plus de collaborateurs, qui sont répartis dans 3 grandes branches :

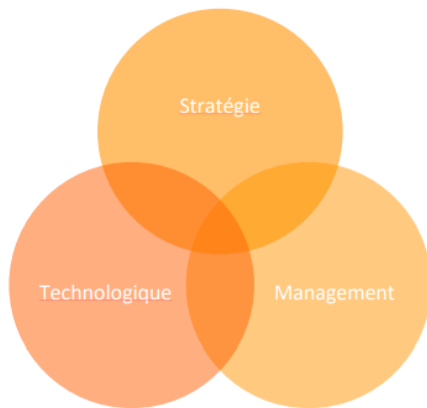


Figure 1 - Cercles d'Aneo

Nous avons la branche stratégie, s'occupant de ce sujet au niveau des organisations, des systèmes d'informations et d'expérience.

La branche management, spécialisée dans le pilotage par la valeur, l'agilité des organisations et la transformation managériale.

Enfin, la branche technologie travaille sur les sujets HPC Cloud de l'entreprise, l'industrialisation des processus logiciels, ainsi que les sujets data et intelligence Artificielle.

Ces 3 grandes branches représentent et font l'essence d'Aneo. Cependant, à l'intérieur de la boîte, nous avons plus l'habitude d'entendre parler de membres « jaunes » et « rouges ». Les derniers regroupent les employés travaillant sur les sujets technologiques. Les jaunes représentent tout le reste. Aujourd'hui, la volonté d'Aneo est de faire les efforts nécessaires pour n'avoir plus qu'un seul groupe, qui serait la fusion des deux actuels. Cela permettrait à tous les collaborateurs d'échanger sur leurs métiers, leurs compétences, et de faciliter leurs relations. Ce groupe serait un groupe « orange », couleur résultante du jaune et du rouge, que l'on retrouve dans le logo d'Aneo.

## 3. Evénements

Aneo organise aussi plusieurs événements. Ceux-ci ont des buts variés : on y retrouve par exemple l'intégration de nouveaux arrivants, la description des actualités de l'entreprise, en passant par des formations inter-collaborateurs.

### A. Aneo Lunch

Le premier de ces événements a lieu les vendredis, une semaine sur deux. Tous les collaborateurs d'Aneo sont invités à y participer, au moins en visioconférence. L'Aneo Lunch se déroule en deux phases :

Lors de la première, certains membres de l'équipe exécutive prennent la parole pour partager certaines évolutions de l'entreprise. Cela commence par la présentation des nouveaux membres de l'entreprise, qu'ils soient stagiaires, alternants, ou employés. Ceux-ci se présentent alors à l'ensemble des personnes présentes.

Pour faire suite, l'équipe RSE de l'entreprise explique les différentes évolutions mises en place par Aneo. Je pourrais citer la fresque du climat et la fresque du numérique, qui sont deux sujets importants pour la sensibilisation à l'écologie, qui a été un sujet souvent ramené sur la table lors de ces occasions.

Le sujet suivant parle des nouvelles affaires d'Aneo, des nouveaux contrats ainsi que des collaborateurs amenés à travailler dessus. Une légère description de la mission est faite pour que chaque collaborateur comprenne exactement en quoi elle consiste. C'est encore une fois dans l'optique de partager les expériences le plus possible entre collaborateurs que ceci est fait.

Enfin, pour clôturer cette première partie du rituel, les prochains évènements internes à la boîte sont présentés, qu'ils soient en rapport avec le ski, l'œnologie, ou encore le jeu vidéo !

La deuxième partie de l'Aneo Lunch est juste une phase de repas, où les employés présents sur site peuvent discuter de n'importe quel sujet !

## B. Afterworks

Aneo n'est pas la seule entreprise proposant des Afterworks. Cependant, j'ai pu découvrir ce genre d'évènement avec Aneo, et ce, avant même le début de mon stage. En effet, les afterworks permettent aux futurs collaborateurs, qui ont un contrat signé, ou qui sont encore dans le processus de recrutement de l'entreprise, de rencontrer les différents employés de la boîte. Par la discussion, ceux-ci peuvent non seulement faire la rencontre de leurs futurs collègues, mais aussi se faire une meilleure idée des valeurs d'Aneo. Même les anciens collaborateurs ayant quitté l'entreprise peuvent rejoindre l'afterwork. Personne n'est jamais isolé de ces évènements.

## C. Demi-journées *Knowledge*

Pour finir avec les rituels d'Aneo, je comptais parler du plus singulier. Les demi-journées *Knowledge* ont lieu lors du dernier vendredi de chaque mois. Pour expliquer simplement le principe, il s'agit d'une période de formation sur un sujet choisi par un collaborateur, enseignée par ce collaborateur, et à destination des autres collaborateurs d'Aneo. Les thèmes sont encore une fois très variés : il peut y avoir des introductions à certaines technologies ou des cours de management. Nous pouvons aussi participer à une fresque du climat proposée par la RSE, et bien d'autres choses encore ! Tout dépend de qui fait la formation.

On retrouve encore une fois dans ce rituel la volonté d'Aneo de faire naître ce cercle « orange », où les compétences de chacun sont partagées à tous. N'importe qui voulant découvrir le management peut profiter d'une demi-journée *Knowledge* pour en apprendre les bases, ou se concentrer sur une méthode particulière. De plus, continuer à se former après les études est toujours quelque chose d'intéressant et de plaisant, et ce rituel permet de toujours faire varier les plaisirs !

Cependant, à cause d'une suite d'éléments imprévus, tels que les grèves et les vacances de fin d'année, je n'aurais pu participer qu'à une seule de ces demi-journées *Knowledge*. J'aurais aimé les découvrir davantage, approfondir la manière dont ceux-ci sont organisés, et en profiter pour apprendre davantage.

## 4. Cadre de vie et travail

Aneo ne se résume pas aux évènements qu'elle propose à ses collaborateurs. D'autres éléments entrent en jeu pour mélanger les cercles de couleur, et cela passe par l'organisation



des locaux ainsi qu'un système très similaire de quelque chose qu'on peut retrouver à l'Efrei et dans d'autres écoles d'études supérieures.

### A. Des salles pigmentées

Les locaux d'Aneo situés à Boulogne-Billancourt, sont organisés d'une manière particulière, élargissant les principes de l'open space. Ceux-ci sont organisés en *flex-office*, ce qui signifie tout simplement qu'aucune place n'est réservée à personne. Bien sûr, pour l'équipe exécutive et les employés travaillant sur des données sensibles ont une place attitrée de manière exceptionnelle. Mis à part ceux-ci, n'importe qui peut décider de prendre telle ou telle place, de changer ses habitudes pour une semaine ou pour un pour un mois, et surtout, de se retrouver à côté de personnes travaillant dans un domaine différent du nôtre. On retrouve encore une fois ce mélange fort présent dans l'identité d'Aneo.



Figure 2 - Flex Office

Les couleurs font aussi partie de l'identité d'Aneo, et les salles de réunions présentes dans les locaux le rappellent aussi. Tout comme les amphithéâtres du bâtiment E de l'école, chaque salle de réunion est associée à une couleur. Deux d'entre elles sont totalement adaptées aux visioconférences, même s'il est tout à fait possible d'en organiser de manière moins efficace dans les autres.

### B. Clubs

Pour finir avec l'intégration des collaborateurs dans l'entreprise, je souhaite parler des clubs de celle-ci. Tout comme les associations de l'Efrei, les clubs proposent des activités en rapport avec une passion. Les employés appartenant à différents cercles sont donc encore une fois mélangés entre eux, par l'intermédiaire d'une passion commune. Parmi ces clubs, on retrouve évidemment le jeu vidéo, mais aussi l'œnologie, l'escalade, et même la zythologie.

Étant très impliqué dans la vie associative de l'Efrei, j'ai été très agréablement surpris par l'existence de ces clubs, que je croyais ne jamais retrouver dans le monde professionnel. Bien que je sois sûr qu'il ne s'agit pas de quelque chose de présent dans chaque entreprise, je suis très content de cette découverte !

## II- ArmoniK

### 1. Présentation du projet

Comme précisé plus tôt dans l'introduction, j'ai pu travailler sur le projet ArmoniK pendant mon stage, et plus précisément sur l'interface administrateur de celui-ci.

ArmoniK est un orchestrateur HTC mis en production en juin 2022, donc très récemment. Il est issu d'un projet plus ancien, nommé HTC-Grid, initialement financé par Amazon Web Services (AWS). Un orchestrateur se base sur le principe de l'orchestration informatique, qui décrit le processus d'automatisation d'organisation, de coordination, de gestion de systèmes informatiques complexes, de middleware et de services. Le HTC, tiré de l'anglais « *High Throughput Computing* », traduisible en « Calcul à Haut Débit », est une technique permettant de calculer un très grand nombre de tâches indépendantes. Le cas le plus connu et représentatif du HTC se retrouve dans la méthode de Monte-Carlo.

ArmoniK permet, entre autre, la gestion et la distribution de calculs de manière parallélisée sur un environnement de *cloud computing*. Cela signifie que les calculs sont distribués à travers des ressources en cloud grâce à un algorithme complexe, de manière à être traités le plus efficacement et rapidement possible. Aujourd'hui seul le cloud AWS est supporté mais dans un futur proche, l'orchestrateur sera disponible et fonctionnel sur différents services de cloud grâce à son architecture hétérogène. Cet orchestrateur a été conçu de tel sorte à pouvoir correspondre à la définition de multicloud, c'est-à-dire adapté à tous les services de cloud.

Il s'agit d'un projet open-source gratuit basé sur *Kubernetes*, aujourd'hui financé par Aneo et les banques l'utilisant. Il est financé en grande partie par le support des développeurs, qui implémentent des nouvelles fonctionnalités ou modifications demandées par les clients. ArmoniK possède bien sûr des concurrents. Dans le monde de la banque existe notamment *IBM Spectrum Symphony* ou *Tibco gridserver*. A la différence de ceux-ci, ArmoniK se veut open source, et adaptable suivant les besoins spécifiques des clients, tout en gardant une très forte performance de calcul et en étant le plus adapté possible au multicloud.

ArmoniK a intégré un système d'interface administrateur, permettant de visualiser et gérer plus facilement les différentes tâches et résultats qu'il renvoie. J'ai pu rejoindre ce projet dans le cadre d'une refactorisation de cette partie de l'application après des demandes clients.

### 2. Ma mission

J'ai donc rejoint l'équipe qui travaillait sur l'interface administrateur d'ArmoniK (GUI). Suite à la mise en production, les clients utilisant l'interface ont fait différents retours nécessitant une refactorisation du code de l'application web<sup>1</sup>. De plus, au départ du projet, l'interface administrateur utilisait sa propre API<sup>2</sup> pour réaliser les différentes récupérations de données directement à partir de la base de donnée MongoDB. Cependant cette API, étant donné qu'une autre partie du logiciel avait la même responsabilité qu'elle, la base de données pouvait être surchargée de requêtes. Elle a donc été supprimée pour utiliser les API gRPC fournies par ArmoniK. Cela a été possible par l'extension de l'API originelle d'ArmoniK pour contenir les

besoins spécifiques de la GUI. Par conséquent, je suis rentré dans un projet sorti récemment, en plein changement, et donc nécessitant beaucoup de travail.

### 3. Équipe et management

Un projet aussi complexe que ArmoniK ne peut pas évoluer sans l'aide d'une équipe dévouée et motivée. La réalisation d'un tel orchestrateur demande un temps important, et doit prendre en compte les demandes clients. De ce fait, l'équipe doit être organisée du mieux possible pour éviter quelconque problème de management. Cela se passe par la mise en place de divers outils et méthodes, que je m'apprête à présenter.

#### A. Agilité ou lévitation ?

Aujourd'hui, la méthode scrum est la plus à la vogue dans les entreprises. Pour simplifier sa définition le plus possible, il s'agit d'une méthode de gestion de projet agile qui consiste à mettre en place des processus d'une durée d'environ 2 semaines appelés « sprints ». Durant cette période, le scrum master et les développeurs vont analyser les besoins du client, puis travailler sur les fonctionnalités demandées, qu'ils doivent compléter d'ici la fin du sprint. Ces fonctionnalités sont alors mises en production. Ensuite, le « product owner » (PO) de l'équipe va récupérer l'avis des clients sur lesquels se baseront les développeurs pour le sprint suivant.

Ce n'est cependant pas la méthode agile choisie par l'équipe d'ArmoniK. Le scrum part du principe qu'un projet informatique avance mieux avec des deadlines courtes. Cela permet au client de rapidement donner son avis sur des ajouts de code. Cependant, les développeurs sont mis sous pression, une pression constante. Du point de vue de l'équipe d'ArmoniK et de ses managers, cela revient à mettre en place une épée de Damoclès au-dessus de la tête des développeurs. De plus, si une pression est constante, elle devient la norme, et elle perd de ce fait tout son intérêt. C'était pour eux un trop puissant contre-argument à la méthode scrum pour pouvoir l'appliquer sur le long terme.

En se basant sur les compétences et les expériences précédentes en méthode agile de certains des membres de l'équipe, les managers et l'équipe d'ArmoniK ont mis au point leur propre modèle de méthode agile. A la place du PO de la méthode scrum, on va retrouver un tech lead en relation avec le ou les clients d'ArmoniK. Mais cette fois-ci, pas de pression, pas de sprint : les livraisons de code sont faites tous les mois et seulement si la qualité du produit est considérée comme acceptable. Si elle ne l'est pas, le rendu peut être décalé de quelques jours, voire semaines.

Pour visualiser plus facilement le moment où une mise en production peut être faite, l'équipe ArmoniK utilise la méthode *Kanban*, permettant de visualiser l'avancement d'un ensemble de tâches. De cette manière, les développeurs font de leur mieux pour réaliser une tâche au plus tôt, sans pour autant avoir la pression d'une *deadline*. Ce n'est cependant pas un processus facile à mettre en place, étant donné qu'elle nécessite d'avoir de nombreux rituels et une confiance presque aveugle entre membres de l'équipe.

## B. La confiance comme fondation

Une équipe où les membres n'ont pas de réelle confiance entre eux va éprouver des difficultés à avancer. Plutôt que de demander une modification du code, un membre va aller le faire lui-même. Un autre va toujours repasser sur le travail d'un autre, en changeant l'entièreté du travail effectué. Tout ceci, en plus d'instaurer une atmosphère tendue dans l'équipe, fait perdre du temps à l'équipe, qui peut être très précieux.

Afin d'éviter tous ces problèmes, il est nécessaire que chaque membre fasse confiance à l'ensemble de l'équipe. Avec le modèle agile mis en place, cela devenait encore plus important. Sinon, en un mois, l'avancée des ajouts peut être minime. Mon maître de stage m'a parlé de la difficulté de mettre en place une telle confiance dans une équipe où les membres ont un grand écart générationnel. Avec plus de 30 ans de différence entre le plus jeune et le plus ancien membre, on comprend cette difficulté. Il peut y avoir des volontés d'imposer ses solutions en tant que senior, ou à l'inverse, à trouver ces solutions vieillissantes en tant que junior. Cependant, même s'il a pris du temps, le défi a été relevé et en observant l'équipe, on se rend très vite compte de l'ambiance positive qui y règne. Personne ne se marche sur les pieds, et tous les développeurs osent demander de l'aide sur un blocage.

En intégrant l'équipe, j'ai pu aussi observer à quel point la stratégie de la hiérarchie plate était intégrée dans les valeurs d'Aneo. Bien que n'étant qu'un stagiaire, mon travail était traité de la même manière que tout le monde. De même pour mes prises de paroles. J'ai toujours pu exprimer mon avis sur les choses sans risquer quoi que ce soit.

## C. Au quotidien

Même si la confiance règne, il est important de faire un point chaque jour sur l'avancement du projet. En mettant en place des réunions appelées « Daily », les développeurs de l'équipe peuvent communiquer chaque jour sur leurs avancements de la veille, en utilisant les cartes de la méthode Kanban pour mesurer leur avancement. C'est le moment pour ceux-ci de demander l'avis de leurs collègues si jamais des problèmes sont rencontrés, et de faire le point sur les sujets les plus urgents. Ce sont des réunions dont la durée est déterminée par la taille de l'équipe, mais qui restent très courtes. Pour ArmoniK, au moment de mon stage, celles-ci ont lieu à 9h30, pour une durée de 30 minutes ce qui est considéré long par la méthode Scrum.

Pour les réunions nécessitant davantage d'informations ou pour préparer des communications avec le client, les « Bi-Weekly » vont être préférés. Au nombre de deux par semaine, ils remplacent les Daily. Ceux-ci sont généralement un peu plus longs, car ils permettent de faire un point un peu plus complet sur les prochaines évolutions à apporter au projet. Si nécessaire, plus de discussions sont faites sur les sujets moins importants de la prochaine livraison de code.

Vers la fin d'année 2022, les Daily ont disparu de l'équipe ArmoniK, notamment parce que la majorité des membres étaient en vacances, travaillaient sur d'autres sujets de l'entreprise, et que le projet ne nécessitait plus autant d'investissement par rapport aux besoins du client, lui aussi en vacances. Ils sont cependant réapparus au début de janvier, dans une période où l'équipe grandissait.

## D. Rétrospection

Un des plus grands problèmes dans la gestion d'une équipe, c'est la stagnation. Une équipe est de taille variable et les gens qui la composent évoluent. Elle aussi doit prendre en compte les besoins des clients, qui sont destinés à être plus ou moins nombreux. Il est normal que les besoins de ces équipes changent, que de nouveaux problèmes naissent dans son organisation. Il est nécessaire d'adresser ces problèmes de temps en temps, et surtout de leur trouver une solution.

Les « Rétrospectives ArmoniK » sont justement là pour répondre à ces nécessités. Cette réunion particulière d'une durée dépassant les 2 heures ne traite aucunement de technique, mais uniquement de la direction que prend l'équipe par rapport aux membres et à la manière de développer. Elle a vocation à améliorer chaque point en rapport avec l'équipe, sans s'occuper une seule seconde des lignes de codes présentes dans le projet. C'est un rituel adapté aux méthodes agiles proposé par Norman Kerth, et repris par Esther Derby et Diana Larsen au début des années 2000.

De nombreux outils ont été créés au fil du temps pour mener à bien cette réunion dans différentes équipes : la montgolfière, le gladiator et le SpeedBoat, pour n'en citer que trois d'entre eux. C'est ce dernier qui est utilisé par l'équipe ArmoniK lors des rétrospectives. Le SpeedBoat est une analogie entre l'équipe et son environnement de travail. L'outil met en place un bateau, représentant l'équipe, qui cherche à atteindre une île. Au-dessus de celui-ci brille un soleil chaleureux, et le bateau, un voilier, est poussé par des vents. Cependant, le bateau est ralenti par son ancre encore baissée et certains récifs l'empêchent de rejoindre l'île tranquillement. L'analogie à laquelle je faisais référence apparaît très clairement : le bateau représente l'équipe, le soleil ses succès, le vent ses forces, l'ancre les freins à son efficacité, et les récifs sont les obstacles à surmonter. L'île, quant à elle, est l'ensemble des objectifs de l'équipe et du projet.

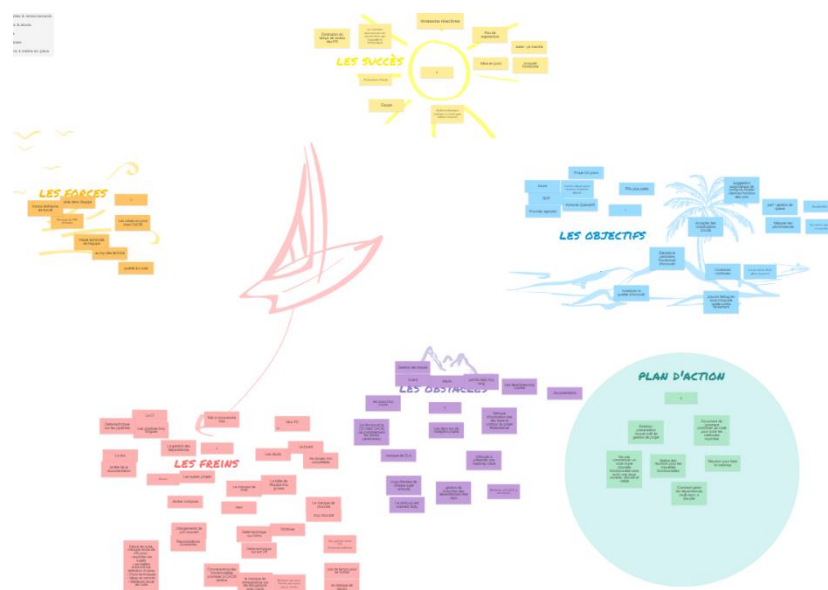


Figure 3 - SpeedBoat d'une Rétrospective ArmoniK

Rapporter un projet de n'importe quel type à un voyage ou à une aventure est intéressant. Il y a toujours une fin, une destination, mais tout ne se passe pas calmement. Il faut savoir analyser les situations dans lesquelles nous nous trouvons et agir en conséquence. Remonter l'ancre quand nécessaire. Prendre conscience des récifs au loin pour trouver la route la plus facile à traverser. Même le plus petit d'entre eux peut poser des problèmes immenses en cas de collision. Le vent est une force dont on ne peut se soustraire, il faut donc apprendre à travailler avec lui.

Pour revenir sur la réunion en elle-même, celle-ci se découpe en trois parties. Tout se déroule sur un plateau (numérique, dans le cas d'ArmoniK, avec la plateforme Klaxoon) ou sont placés les différents éléments du SpeedBoat. L'animateur du rituel va d'abord laisser les membres de l'équipe poser des cartes sur les différentes zones du plateau. Chacune de ces cartes correspond à un accomplissement, un objectif, un frein... Par exemple, dans « les forces », nous pouvons retrouver une carte « qualité du code », dans « succès » ce sera « livraisons réactives »... Dans un deuxième temps, l'ensemble de l'équipe va trier les cartes ensemble pour essayer de les regrouper par catégories. C'est aussi un moment de discussion autour de ces cartes, pour savoir si leur place sur le plateau est légitime ou non. Grâce à cela, l'équipe va pouvoir voir les succès ou les problèmes qui ont des incidences entre eux. Ainsi, il est plus facile de trouver une unique solution pour ces problèmes, ce qui simplifie le temps de travail et l'implication nécessaire pour les corriger. Enfin, l'équipe va pouvoir s'attaquer au plan d'action. Celui-ci consiste à trouver les solutions à mettre en place afin de se débarrasser du plus de problèmes possibles de la manière la plus efficace possible. Lors de certaines Rétrospectives, ces solutions peuvent se trouver peu nombreuses, mais l'équipe s'attend alors à ce qu'elles aient un grand impact sur leur environnement et les habitudes de ses membres.

Il s'agit donc d'un outil qui permet à chacun de d'abord donner son avis sur différents aspects de l'organisation, et ensuite de discuter et de débattre sur ceux-ci. L'Homme étant un animal sociable, un outil basé sur la discussion semble totalement adapté pour l'aider à évoluer facilement. Organisées suffisamment fréquemment, ce genre de réunions permet à l'équipe de se débarrasser de nombreux problèmes gênants et permet d'éviter que de nouveaux apparaissent trop vite.

Il n'est pas rare lors de ces séances de remarquer que plus de freins et d'obstacles ont été trouvés que de succès ou de force. Ce n'est pas quelque chose d'étonnant, l'être humain a plutôt tendance à relever le négatif plutôt que le positif. Cependant, comme expliqué plus tôt, les cartes sont regroupées entre elles, et on remarque alors que le nombre de problèmes ayant le même thème et le nombre de succès et de forces sont relativement similaires. De ce fait, ces derniers sont remis en avant, et cela permet à l'équipe de se rendre compte de l'importance de ses réussites, tout en cherchant des solutions à ses problèmes.

Bien qu'ayant entendu parler des méthodes agiles et de certains outils de management pendant mon cursus à l'Efrei, je n'ai pas souvenir d'avoir entendu une allusion aux Rétrospectives. C'est un rituel qui m'a convaincu par son principe fondé sur la discussion. Il est tout à fait adapté dans une équipe où la confiance règne, et où l'ancienneté ne donne pas plus de pouvoir. Il serait intéressant de le comparer à d'autres rituels mis en place dans d'autres entreprises, ou de voir les différentes manières de l'utiliser.



## E. Choisir la direction

Pour entamer la transition entre gestion d'équipe et technique, je compte parler d'un rituel un peu moins commun que la rétrospective, qui, cette fois-ci, s'occupe uniquement du projet et de la direction que celui-ci prend. Cette cérémonie consiste à mettre en place une *Roadmap*. Cette réunion a pour but de trouver les différents besoins qui feront évoluer le projet positivement. Ces besoins peuvent être liés à des nécessités techniques, comme à des outils permettant de vendre le logiciel. Les développeurs vont donc fournir une liste de ce qu'ils pensent être important, toujours en discutant, puis vont prioriser cette dernière. De cette manière, l'équipe décide de la direction à prendre, et chacun de ses membres sait ce qui doit être réalisé en premier.

Etant donné que ArmoniK est un projet encore jeune, ces *Roadmaps* ont une importance capitale pour le faire évoluer dans le bon sens. Les décisions sur la technique peuvent avoir une forte incidence sur le long terme, notamment au niveau des dettes techniques<sup>3</sup>. Il est important d'organiser une telle réunion lorsque les priorités commencent à manquer.

## 4. Intégration de Git

### A. Présentation

Git est un outil de versionnage de code. Il permet de facilement travailler sur un projet informatique, en fournissant une version différente de celui-ci sans risquer d'intégrer du code cassant. Grâce à un système de branches, de merge et de pull request, git permet de travailler sur de nouvelles fonctionnalités tout en maintenant la sécurité et le bon fonctionnement de celui-ci. D'abord spécialisé dans la gestion de code, certaines de ses fonctionnalités sont vouées à être utilisées dans une approche « projet ».

### Branches

Lorsque l'on veut faire une modification sans pour autant risquer de casser le code utilisé par des clients, et donc sans risquer de le rendre inopérant, nous pouvons créer une nouvelle branche git. C'est un concept qui porte bien son nom. Le code est un arbre, et nous souhaitons faire pousser une branche correspondant à un changement. Avant de l'ajouter, nous devrions vérifier si sa fusion ne détruirait pas d'autres branches déjà présentes sur l'arbre, voire la destruction entière de ce dernier.

Les branches peuvent être partagées sur le dépôt Git, si bien que d'autres contributeurs peuvent la modifier. Les modifications que chaque contributeur apporte sont appelées « *commit* ». Une branche va donc représenter un ou plusieurs commits, pouvant être poussés avec des « *push* » et sont récupérables avec des « *pulls* » par tous les contributeurs.

Aujourd'hui, dans la plupart des cas, les dépôts git sont organisés autour des branches. Nous allons généralement avoir une branche principale, qui est totalement fonctionnelle et fournie aux clients ou utilisateurs de l'application. Les autres branches correspondent chacune à une modification, qui, selon la méthode git employée, sont de tailles très variables.

Après que le développeur travaillant sur une branche pense que toutes les modifications apportées sont correctes et ne créent aucun bug, il peut passer à l'étape suivante, la fusion, à l'aide d'une *Pull Request*.

### Merge et Pull Request

Fusionner une branche sans vérification préalable du code n'est ni une pratique recommandée, ni dans la continuité de la logique d'utilisation des branches. Même avec quelqu'un digne de confiance ou plein de bonne volonté, qui a testé le code de son côté, des erreurs peuvent toujours être présentes. Il faut donc :

- Tester le code.
- Relire les modifications.

Généralement, nous faisons appel à un voire deux relecteurs. Leur mission est d'assurer que la fusion de la branche modifiée vers la branche principale ne va pas casser quelque fonctionnalité, voire toutes les fonctionnalités. Même si un code est très correct, des problèmes peuvent subsister, et ce à cause de l'utilisation de certaines méthodes git dans une équipe.

Prenons le cas où deux personnes créent leurs modifications en se basant sur la même version de la branche principale. Si celles-ci modifient des fonctionnalités différentes et donc des lignes de code différentes, et que l'une d'elle retrouve sa branche fusionnée à la principale plus tôt que l'autre, il n'y aura aucun problème de compatibilité. Cependant, si ces deux personnes travaillent sur les mêmes lignes de codes (pour des raisons différentes), lors de la fusion de la deuxième branche, il risque d'y avoir un écrasement du code de la première personne et l'une, l'autre, ou les deux fonctionnalités apportées ou modifiées par les branches ne pourraient alors fonctionner, faute de compatibilité.

Il faut donc faire attention à toujours garder les branches à jour avant de fusionner le code, en faisant des « *rebases* », qui permettent de prendre les commits d'une autre branche et les ajouter aux nôtres, en indiquant les endroits où il y a conflit de modifications. Ces conflits peuvent alors être réglés intelligemment par le développeur ou les relecteurs.

La relecture du code peut sembler laborieuse. Cependant, git possède des outils bien pensés, et les *Pull Requests* (PR) ne font pas exception. Plutôt que de laisser les relecteurs regarder le code directement dans les branches, les *Pull Requests* permettent de visualiser exactement les endroits où les lignes ont été modifiées, et de les comparer avec celles de la branche principale. De cette manière, nous pouvons facilement voir les modifications dans chaque fichier et vérifier le code beaucoup plus facilement. Chaque relecteur peut lancer des conversations sur une ou plusieurs lignes de code, faire une suggestion de changement, poser des questions et bien entendu approuver ou non ces *Pull Requests*. Après avoir été approuvées, la branche peut être fusionnée avec la branche principale. En d'autres termes, elle peut être rajoutée à notre arbre.

Les *Pull Requests* peuvent être utilisées pour la fusion de n'importe quelles branches, qu'elles soient principales ou non. Il y a cependant un intérêt moindre à les utiliser pour la fusion de toutes les branches. C'est pour cela que git propose de protéger les branches de modifications apportées sans *Pull Request*.

### Historique et guidelines



La relecture de code est grandement simplifiée grâce aux outils de git, mais il reste encore une part d'efforts que le développeur souhaitant ajouter des changements doit fournir. Ces efforts ne touchent pas seulement les *Pull Request*, mais sont aussi très importants pour l'historique git.

Ce dernier représente la liste de tous les changements apportés au projet. Tout y est noté depuis le premier *commit*, jusqu'au plus récent *merge*. Certains schémas que j'utilise dans cette partie ne sont qu'une visualisation d'un historique git. Il permet à n'importe quel développeur de savoir quelles sont les fonctionnalités implémentées et quand elles l'ont été. De plus, il permet de fournir une liste des changements appliqués aux clients depuis la dernière mise à jour du projet.

Il est donc impératif d'avoir un historique git clair et concis, et cela passe par plusieurs étapes. Un développeur ne peut pas arriver avec un commit correspondant à divers changements et fonctionnalités. Cela peut très bien se faire dans certaines *Pull Request*, mais pas dans un *commit*. Le *stage*, étape précédent un *push*, est un outil permettant de diviser un changement en plusieurs *commits*. Il permet de sélectionner les changements que nous souhaitons apporter au *commit*. Chaque apport de code est donc totalement séparé, rendant l'historique plus clair.

De plus, les *commits* doivent avoir un nom clair et concis pour être compréhensibles rapidement pour quiconque les lit. Généralement, la première ligne de leur nom ne doit pas dépasser 50 caractères, mais elle peut être suivie d'une explication plus longue. Certains projets suivent une convention de nommage. Elles sont généralement dépendantes de la technologie utilisée dans le projet. C'est le cas de l'interface Administrateur d'Armonik sur laquelle j'ai travaillé. Elle suit la convention de *commits* d'Angular (voir sources), notamment sur la nature des changements. S'ils représentent de nouvelles fonctionnalités, des corvées, une réparation, ils seront nommés différemment, encore dans l'optique de simplifier leur lisibilité.

Mais avoir trop de commit peut aussi être un problème dans l'historique. En effet, si des modifications sont demandées et appliquées sur une *Pull Request*, le nombre de *commit* peut rapidement augmenter et donc pourrir l'historique. Cela m'est d'ailleurs arrivé, au début de mon *stage*, avec une *Pull Request* composée de 101 *commits*, quand je ne connaissais encore aucune des conventions que j'ai cité plus tôt. Pour éviter de salir l'historique Git, avant de fusionner une telle *PR*, nous pouvons *squasher* nos changements, c'est-à-dire regrouper tous ceux-ci en un unique commit.

Si le développeur a correctement respecté les conventions, cela ne peut qu'améliorer l'état de l'historique. Un *commit*, tout comme une *PR*, peut correspondre à une fonctionnalité ou à une modification. Alors en principe, une *Pull Request* peut être vue comme un *commit* et le *squash* garde tout son sens. Maintenant, il suffit de suivre la convention de nommage des *commits* et l'appliquer aux *Pull Requests* avant le *squash* pour que l'historique Git reste propre.

## B. Utilisation de GitHub

Maintenant que nous avons une bonne idée du fonctionnement de Git, il est temps de parler de la plateforme GitHub. Il s'agit d'une plateforme d'hébergement de git créée en 2008 et rachetée en 2018 par Microsoft. Dans son principe de base, GitHub est pour Git ce que YouTube ou Dailymotion sont à la vidéo. La plateforme ne s'est pas contentée de ça, mais a su

proposer des outils complémentaires à Git dans le cadre de la gestion de projet. Git permet de simplifier la vie des développeurs au niveau du code, et GitHub permet d'avoir une vue d'ensemble sur le projet ainsi que sur ses possibles évolutions.

GitHub héberge des dépôts Git, communément nommés *repositories* (diminué en *repos*). Git permet à n'importe qui en ayant l'autorisation de développer sur un projet hébergé sur un serveur. GitHub (et GitLab, concurrent de la plateforme) est une solution permettant d'héberger un *repo* git de cette manière. Le propriétaire de ce dernier peut alors décider de laisser son projet privé, le laissant accessible uniquement aux utilisateurs qu'il a invités. Il peut aussi l'ouvrir au public, permettant à littéralement tout le monde de lire et étudier le code. C'est ce que l'on appelle de l'*Open-source*. Même sur un projet public, des conditions peuvent être créées pour empêcher n'importe qui d'ajouter ses modifications sans vérification préalable du propriétaire. GitHub permet donc d'implémenter un système d'autorisation et de visibilité aux *repos* git.

## GitLab

Avant de commencer à décrire les diverses fonctionnalités de GitHub, j'aimerais parler de son principal concurrent, GitLab. C'est une plateforme très similaire, possédant de nombreux points communs avec GitHub. Les différences se feront ressentir dans l'intégration de certaines fonctionnalités, qui relèvent parfois de l'ordre de la nuance. Cependant, chaque plateforme a son avantage sur l'autre, si bien que choisir l'une ou l'autre est un dilemme complexe. Cependant, trouver une solution complexe pour un problème comme celui-ci dans le cadre d'une gestion de projet peut prendre du temps. Il est donc préférable de d'abord choisir l'option de la simplicité, et de voir si un changement est souhaitable au fur et à mesure du temps. ArmoniK a donc été hébergé sur GitHub car HTC-Grid, projet prédécesseur d'ArmoniK, était lui aussi sur cette plateforme.

## Issues & Discussions

Sans doute un des outils les plus utilisés par les utilisateurs de GitHub, les *issues* sont presque nécessaires pour faire évoluer un projet. *Issue* signifie « problème » en anglais, et c'est encore une fois un nom très révélateur du concept auquel il est rattaché, mais pas autant que c'était le cas pour les branches de git.

Dans le cadre d'un projet open-source, les *issues* peuvent être utilisées par n'importe quel utilisateur de GitHub pour décrire un souci que ce dernier rencontre dans le code, ou dans l'application. Généralement, l'utilisateur va fournir toutes les étapes qui permettent de recréer le comportement du problème. Les développeurs travaillant sur le projet vont l'aiguiller sur les diverses solutions pour résoudre l'*issue*, et appliquer un correctif de bug via une *PR* si besoin. Il n'est pas rare que les développeurs du projet en créent eux-mêmes, en trouvant de nouveaux bugs alors qu'ils modifiaient le code.

Les *issues* peuvent aussi être utilisées pour demander de nouvelles fonctionnalités. Dans ce cas-ci, elles sont plus représentatives de tâches à accomplir. Elles sont très généralement ajoutées par les développeurs à la suite de demandes clients ou par nécessité d'implémenter une nouvelle fonctionnalité.

De plus, GitHub permet de créer des modèles (aka *templates*) d'*issues*. Cela permet de faciliter la création de ces dernières et permet aux développeurs de mieux comprendre le contexte de

ce problème. Voici comment les templates créés par les développeurs sont présentés à un utilisateur dans GitHub :

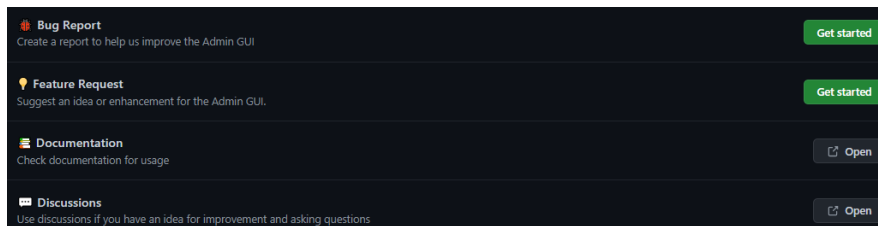


Figure 4 - Modèle d'issues

Cette image est un exemple tiré du dépôt GitHub de l'interface administrateur d'ArmoniK. De nombreux autres modèles peuvent être créés, notamment sur des projets de plus grosses entreprises.

Les *issues* sont ensuite reliées à une *PR*. Cette dernière, quand elle sera acceptée, fusionnée et fermée, fermera automatiquement l'*issue* liée. Celle-ci ne sera pas supprimée, mais invisible tant que certains filtres n'ont pas été appliqués.

Mais j'aimerais attirer votre attention sur le dernier lien proposé dans l'image des templates, les « discussions ». Celles-ci sont utilisées, comme décrit sur l'image, pour poser des questions, proposer de nouvelles features sans pour autant leur coller l'étiquette de « tâche » reliée implicitement à n'importe quelle *issue*. Quelqu'un peut proposer une manière d'implémenter une nouvelle fonctionnalité dans une discussion avec les développeurs, et si jamais aucun compromis n'est trouvé, la discussion sera tout simplement fermée. Dans le cas contraire, une *issue* pourra être ouverte. Les discussions peuvent aussi être utilisées pour poser des questions sur l'utilisation du code dans un autre projet, lorsque l'un de ces utilisateur rencontre un problème d'intégration.

## Milestones

Un autre outil apporté avec GitHub, c'est le jalon, aka *Milestone* en anglais. Comme dans une course, elle représente une borne à atteindre. Elle est composée de nombreuses *issues*, qui correspondent chacune à une étape de développement. Pour ArmoniK, elle est utilisée pour les livraisons de versions.

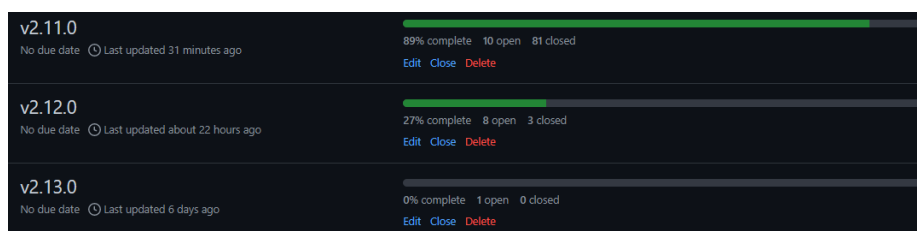


Figure 5 - Milestones GitHub

Les *milestones* permettent de visualiser facilement l'avancement d'une version du projet. Même si elles ne sont pas nécessaires, c'est un outil très adapté pour mettre de l'ordre sur les *issues*, et donc les priorités. En effet, ce sont majoritairement les *issues* correspondant à la *milestone* la plus vieille qui doivent être implémentées en premier.

## Projets

Une autre manière d'organiser les *issues* GitHub, c'est l'utilisation de l'outil intégré *Projet*. Il permet d'organiser simplement les issues dans différents tableaux. Celles-ci deviennent des cartes qu'on déplace dans chaque tableau en fonction de son avancement. On a donc les tableaux *ToDo*, *en cours*, *revue en cours*... C'est une solution très similaire à Trello ou à Monday, mais on voit l'avantage d'utiliser les *Projets* github plutôt que d'autres solutions dans le cadre d'un projet comme ArmoniK, ne serait-ce que pour l'intégration des *issues* en tant que cartes.

Les tableaux permettent aussi d'organiser les *issues* entre elles. En effet, une *issue* GitHub peut en contenir plusieurs, et de ce fait nous pouvons décomposer de la manière la plus atomique possible chaque fonctionnalité. Dans le cadre d'ArmoniK, les *issues* les plus générales possèdent l'étiquette *EPIC* révélant l'importance celles-ci. Elles sont généralement composées d'une dizaine d'*issues* différentes à la complexité variée.

Il y a cependant quelques soucis d'intégration des *issues* dans les *projets* GitHub. En effet, lorsqu'une *issue* est liée à une *Pull Request*, ou qu'elle est fermée, on s'attend à ce qu'elle soit déplacée automatiquement dans les tableaux correspondant dans projets. Ce n'est pas le cas, et les développeurs doivent déplacer à la main les *issues* dans les bons tableaux, en fonction de leur état. Pour certains, c'est un avantage, car cela permet de faire un point sur l'avancement des tâches lors des Daily ou Bi-Weekly sans risquer de voir une carte disparaître sans demander son reste.

Un membre de l'équipe d'ArmoniK a proposé un nouvel outil automatisant le déplacement des cartes d'un état à un autre. Cet outil, Volta, peut être ajouté à un *repo* git pour remplacer l'interface projet. Ce n'est pas une application interne de GitHub, mais elle a pour vocation d'améliorer le traitement des *issues* dans un *repo* git. Elle est capable de détecter, en fonction de l'état de la *PR* reliée à une *issue*, dans quel tableau placer la carte. Par exemple, si aucune *Pull Request* n'est liée à la carte, celle-ci sera en *Next To Start*. Si la *Pull Request* peut être relue, elle sera placée dans *Review In Progress* automatique par Volta. Cet outil permet donc de voir quelles sont les tâches à accomplir dans la journée, sans avoir à se préoccuper à changer l'état de la carte manuellement.

## Workflow

Aujourd'hui, l'utilisation de Git et les principes mis en place autour de cet outil permettent de garantir d'avoir le code le plus fonctionnel possible en production. Ces principes ont de plus en plus tendance à rejoindre l'idée de *l'Intégration Continue* et du *Déploiement Continu*, qui sont des pratiques DevOps je détaillerai un peu plus tard. Dans le cadre de celles-ci, il est important de s'assurer que le code ait bel et bien le comportement souhaité. Par conséquent, les développeurs doivent faire passer des tests au code avant de créer la *Pull Request*. Les tests peuvent être faits sur l'ordinateur du développeur, mais un oubli est toujours possible. Dans ce cas, un bug pourrait entrer en production sur le *repo* git.

Pour suivre la pratique de *l'Intégration Continue*, GitHub a proposé un outil, comme ses concurrents, pour faire passer ces tests à chaque *commit* du développeur. Cet outil est appelé *actions* ou *workflow*. Comme décrit précédemment, il permet de faire passer une batterie de tests, qu'ils touchent à la rédaction du code ou son comportement. Les *actions* permettent aussi d'automatiser des tâches après les vérifications, simplifiant la vie des développeurs. Par exemple, pour l'interface administrateur d'ArmoniK, le code est compilé et envoyé dans une

*image Docker* qui permettra d'assurer le bon fonctionnement de l'application dans n'importe quel environnement. Cette étape supplémentaire rentre d'ailleurs davantage dans le cadre du *Déploiement Continu*.

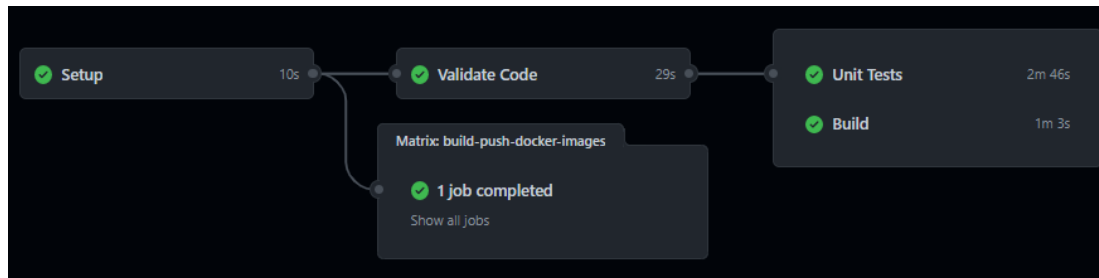


Figure 6 - Visualisation schématique des pipelines GitHub

Les actions sont définissables facilement, tout comme les modèles d'*issues*, directement dans le code du projet dans un dossier « github ». Elles sont définies dans des fichiers « yaml », qui possèdent une syntaxe bien particulière, mais qui sont humainement très lisibles. C'est un outil très intéressant, qui est aussi trouvable chez les concurrents de GitHub, comme GitLab ou Jenkins.

### C. Organisation du projet

#### De nombreuses petites branches

Git, mélangé à GitHub, est un excellent outil de gestion de projet. Cependant, il existe différentes manières d'organiser le dépôt. Ces méthodes ont une influence sur la manière de laquelle les branches sont créées et fusionnées. Parmi elles, nous pouvons retrouver *Git Flow*, la plus ancienne et encore aujourd'hui la plus utilisée, et l'approche *Trunk-based development*, plus récente mais prenant une place de plus en plus importante dans les équipes informatiques.

Commençons avec *Git Flow*. Le principe de cette méthode est de créer une branche par fonctionnalité. Le développeur travaillant sur cette branche va implémenter tous les outils dont il a besoin pour mettre en place cette fonctionnalité. Par conséquent, les branches créées ont une longue durée de vie.

Les branches créées avec la méthode du *Trunk based development*, au contraire, ont une durée de vie très réduite. Avec cette méthode, les branches sont créées dans l'optique d'implémenter la partie la plus atomique possible d'une fonctionnalité. Par exemple, si celle-ci est une calculatrice, il peut y avoir une branche pour implémenter l'addition, une autre pour la soustraction, et ainsi de suite.

Aujourd'hui, la pratique la plus encouragée par les études (notamment DORA, voir sources) est la seconde. Le fait est que la méthode du *Git Flow* pose de gros problèmes au niveau de la fusion de branches. Ces problèmes sont causés par l'existence des branches à durée de vie longue. A chaque fusion de branche de développement sur la branche principale, la probabilité d'avoir des conflits de code, c'est-à-dire une ligne de code modifiée de 2 manières différentes, augmente. Et les branches, vivant longtemps, contiennent de nombreuses modifications du code issu de la branche principale. De ce fait, les cas de conflits sont souvent très nombreux, et prennent énormément de temps à régler. Parfois, le temps qu'il faut pour régler ces

problèmes de conflit peut être suffisamment long pour en apporter de nouveaux. De ce fait, l'efficacité de l'équipe est fortement pénalisée.

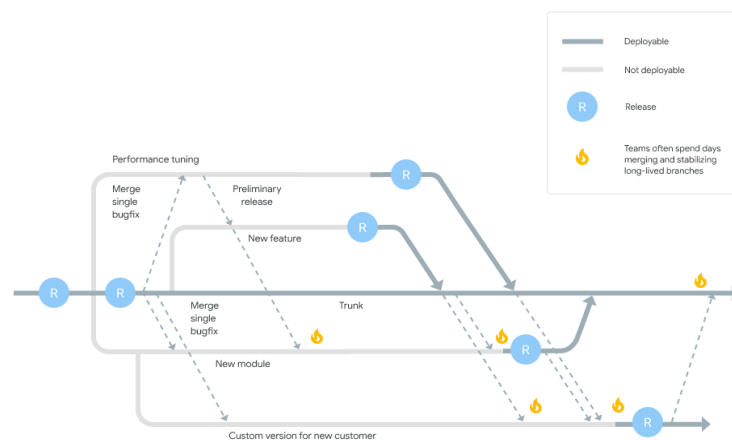


Figure 7 - Historique Git d'un projet réalisé avec la méthode Git Flow

On observe sur ce schéma une visualisation d'un arbre Git dans le cadre de la méthode GitFlow. Ici, on remarque que pendant un très long moment, les branches ne possèdent pas de code fonctionnel (trait gris clair). Qu'à chaque fois qu'une fusion est faite vers une branche non-principale (flèche pointillée descendante), on est souvent confronté à de nombreux conflits. Ces conflits résultent individuellement en une perte de temps de durée variable, mais ils représentent ensemble une énorme quantité d'énergie déployée à les corriger.

Le *Trunk based development* permet d'éviter des cas comme celui-ci. Dans le meilleur des cas, les branches ne vivent pas plus d'une journée, et sont donc fusionnées fréquemment. Il est certes toujours possible d'avoir des conflits, mais ils sont beaucoup moins fréquents et plus faciles à corriger, étant donné que la quantité de code modifié dans la branche est minime.

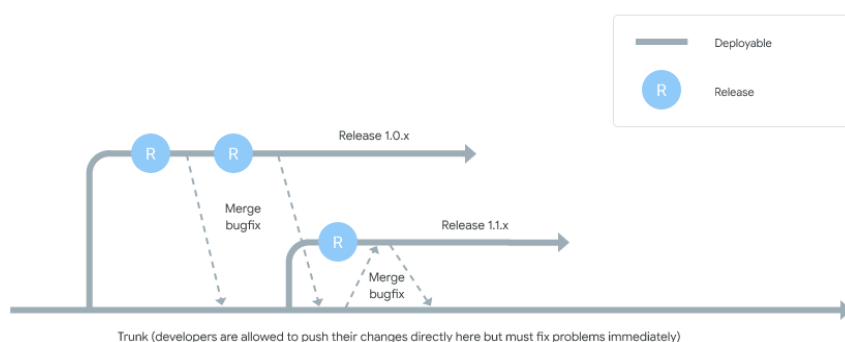


Figure 8 - Historique Git d'un projet réalisé avec la méthode Trunk-Based

La première différence visible avec l'approche *Trunk-Based* est que les branches sont constamment dans un état déployable, c'est-à-dire qu'il n'y a jamais de code cassant. On remarque qu'à chaque modification apportée, cette dernière est directement fusionnée avec la branche principale. Ce schéma montre une approche où la branche, appelée release, est réutilisée après la fusion. Il est cependant possible de la détruire et d'en créer une nouvelle,

comme expliqué plus tôt dans cette partie. On observe aussi qu'il n'y a aucun conflit pour le même nombre de fusions que le code précédent, même si ceux-ci sont encore possibles.

L'équipe ArmoniK a préféré opter pour la seconde solution, notamment en s'appuyant sur les résultats de l'expérience DORA. Le gain de temps et d'efficacité que propose le *Trunk-Based* était bien trop important pour ne pas l'utiliser. De plus, l'expérience indique aussi que la santé mentale des développeurs est moins mise en jeu, car ceux-ci savent que peu de conflits vont apparaître lors de la fusion de branches, et que leur code sera toujours fonctionnel.

### Un arbre à plusieurs troncs

Cependant, l'organisation du projet ArmoniK sur Git ne se limite pas à ça. Le projet contient une grande quantité de code rédigée dans de nombreux langages : C#, python, gRPC, TypeScript... La taille d'un tel dépôt de code ainsi que son organisation aurait été très complexe, et de ce fait, rendrait le projet moins accessible aux nouveaux arrivants. Cela pourrait même, à terme, contribuer à créer une dette technique plus grande, de par sa trop grande complexité auquel personne n'oserait toucher. Et un code complexe signifie plus de difficultés de rendu.

Nous avons remarqué que l'équipe d'ArmoniK aime faire les choses le mieux possible, elle a donc décidé d'éviter de travailler avec un unique dépôt Git. ArmoniK est donc un projet hébergé en « multi-repos ». Cela signifie que plutôt que de regrouper le code à un seul endroit, il va être décomposé en plusieurs sous-repos Git. Cela permet non seulement d'empêcher les problèmes que j'ai cité plus tôt, mais permet aussi aux sous-équipes travaillant sur ce projet de travailler indépendamment les unes des autres. De nombreux outils permettent ensuite « d'exporter » le travail des développeurs et de les faire fonctionner ensemble. Parmi ces outils, nous pouvons retrouver Docker, Terraform et Kubernetes, qui permettent ensemble de déployer et maintenir une infrastructure logicielle. Avec des approches supplémentaires, comme l'*Intégration Continue* et le *Déploiement Continu*, les modifications d'infrastructures sont facilitées.

### D. Initiation au DevOps

Mais alors, qu'est-ce que donc que l'*Intégration Continue* et le *Déploiement continu*, auxquels je fais référence depuis quelque temps maintenant ? Les deux représentent une culture de développement facilitant énormément l'intégration de nouveau code de manière continue dans des productions.

L'*Intégration Continue* (CI) est la pratique qui consiste à implémenter le plus petit morceau de code possible avant de le passer dans un ensemble de processus automatisés de test et de compilation afin de garantir la fonctionnalité de celui-ci. C'est une pratique très adaptée à l'approche *Trunk-Based*, étant donné qu'elle nécessite que le code soit le plus petit possible. De cette manière, les développeurs sont sûrs de la validité présente du code. Sur le long terme, cette pratique a un impact très important sur la qualité d'un logiciel.

Le *Déploiement Continu* (CD) est la suite logique de la CI. Cette pratique consiste à faire en sorte que chaque nouvelle version du code soit configurée et paramétrée de manière à toujours être disponible et fonctionnel en production. Généralement, la CD est mise en place de la même manière que la CI, à l'aide de processus automatiques s'occupant de compiler et de livrer le code de manière continue.



L'intégration de ces pratiques dans un projet comme ArmoniK se fait notamment à l'aide des pipelines GitHub, aka les actions, que j'ai pu présenter plus tôt. Celles-ci, dans le cadre de la CI, vont tester de diverses manières la qualité et la fonctionnalité des changements apportés au code pour vérifier le bon fonctionnement de celui-ci. Ensuite, dans le cadre de la CD, le code sera compilé et envoyé dans un container, outil informatique que je décrirai sous peu.

## Le DevOps

La CI/CD prend tout son sens dans la philosophie du DevOps, terme que j'ai eu maintes fois l'occasion d'entendre et de définir lors de ma scolarité, sans pour autant avoir d'exemples concrets de ce qu'il signifie.

Pour mettre un peu de contexte, auparavant, les développeurs étaient séparés des opérateurs. Ces derniers s'occupent de mettre en production le code rendu par les développeurs, et d'apporter un support aux utilisateurs. A cette époque, les développeurs ne s'occupaient que de produire un code fonctionnel chez eux, et laisser le problème de l'implémentation pour les opérateurs. Ceux-ci, qui ne savaient pas comment le code fonctionnait, essayaient de s'occuper de son déploiement, et notifiaient les développeurs que le code ne fonctionnait pas chez eux. Les développeurs vérifiaient alors le code, qui était fonctionnel, et le renvoyait chez les opérateurs, qui faisaient la même chose... En d'autres termes, le système n'était pas efficace faute de communication entre développeurs et opérateurs.

Le mouvement DevOps est donc né en 2007. Il consistait à former des équipes de développeurs et d'opérateurs, qui travailleraient alors main dans main dans l'optique de livrer efficacement les logiciels. Aujourd'hui, elle a évolué à un point où le développeur est aussi opérateur, et est donc capable d'implémenter un code facile à mettre en production<sup>4</sup>. A l'aide de 8 pratiques déterminées par les principes du DevOps, les équipes ayant adopté cette philosophie sont capables de gérer avec simplicité n'importe quelle livraison.

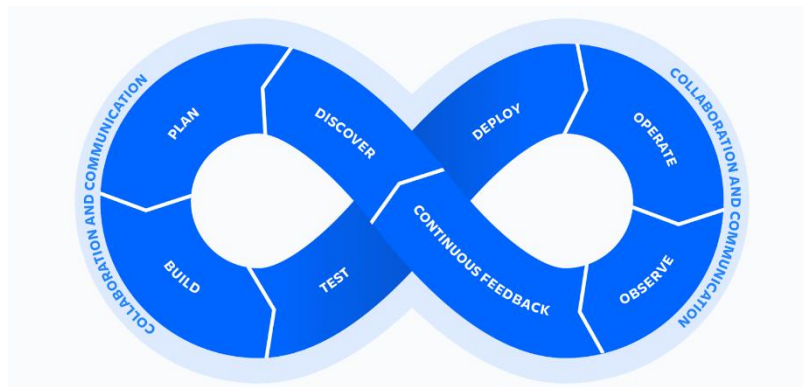


Figure 9 - Cycle DevOps

On peut rassembler les méthodes DevOps en un cycle d'étapes à respecter. Les deux premières étapes nécessitent que les développeurs découvrent les technologies et évolutions nécessaires pour régler un des problèmes découverts lors de la phase de mise en production. Ils doivent ensuite planifier, étape par étape, la manière de laquelle ces modifications doivent être implémentées.

Les deux étapes suivantes sont liées à la pratique CI, étant donné qu'elle nécessite de compiler le code et de le tester. Pour suivre, la CD permet aux équipes DevOps de déployer facilement



le code pour le livrer. Ensuite, l'équipe opère le changement de code dans la production, pour qu'il soit disponible chez les clients de l'entreprise. Après, l'équipe doit observer le comportement du code pour identifier les problèmes survenus, et finir par faire un retour sur l'ensemble des étapes pour améliorer les problèmes lors de la prochaine itération du cycle.

## 5. Déploiement d'ArmoniK

Aujourd'hui, ArmoniK est déployable à la fois sur AWS, en OnPrem et sur sa machine locale. Le déploiement se fait à l'aide de trois technologies : Docker, Kubernetes et Terraform. Elles ont toutes un but différent, mais fonctionnent pourtant efficacement ensemble. Je vais explorer chacune d'entre elles dans les prochaines parties.

### A. Docker, containers, images

Un problème récurrent en informatique, c'est l'environnement. Chaque machine, ordinateur comme serveur, représente un environnement différent. Or, un code fonctionnel sur un environnement peut ne plus l'être sur un autre. C'était d'ailleurs la première raison qui faisait que développeurs et opérateurs ne s'entendaient pas, étant donné que les deux travaillaient généralement dans des environnements différents. La première solution à naître pour éviter ce genre de problème était d'utiliser une Machine Virtuelle (VM) pour héberger le code. De cette manière, on déployait la même VM sur des serveurs et on avait enfin un comportement similaire du code !

Cependant, les VM posent certains problèmes : elles sont dépendantes de l'environnement du serveur, mais surtout, elles demandent de nombreuses ressources. Il faut s'imaginer qu'on déploie plusieurs ordinateurs sur des serveurs avec toutes les fonctionnalités qu'ils contiennent, sans pour autant avoir l'utilité de chacune d'entre elles. Ces fonctionnalités demandent plus de puissance de calcul, plus de ressources. C'était donc une solution fonctionnelle, mais malheureusement peu pérenne.

Fort heureusement, une autre solution, similaire dans l'idée, est née. Il s'agit des *conteneurs*, ou *containers* en anglais. Ils ont été introduits par l'entreprise Docker et ont révolutionné le monde de l'informatique. Un conteneur est tout simplement une VM n'ayant que les fonctionnalités nécessaires pour faire fonctionner un logiciel. Ils sont par conséquent extrêmement légers, en plus d'être facile à créer et à partager.

Les conteneurs sont créés à partir d'*images*. Une image est une sorte de schéma qu'utilise le conteneur pour se créer. Il comporte les instructions exactes pour que le programme puisse démarrer et être accessible par les utilisateurs. N'importe quel développeur peut créer ses propres images, les rendre publiques ou privées. Pour fonctionner, les images se basent généralement sur d'autres images contenant toutes les informations pour exécuter un certain langage de programmation. Par exemple, si l'on veut créer un conteneur faisant fonctionner du code python, nous allons utiliser l'image officielle python.

Les conteneurs Docker utilisent notamment le Docker Engine pour être créés et gérés. Le Docker Engine n'est absolument pas dépendant de l'environnement de la machine sur laquelle il se situe, rendant les conteneurs fonctionnels à 100% sur toutes les machines.

Les différentes parties du code d'ArmoniK (Core, Admin GUI, Api...) séparées dans les différents dépôts git sont toutes exportées en tant qu'images avec leur environnement adapté. Ces images sont ensuite stockées sur *Docker Hub*, qui est une plateforme où stocker et récupérer des images de manière publique ou privée.

## B. Kubernetes

Maintenant que nous avons nos images, qui possèdent exactement les informations nécessaires pour faire fonctionner les différentes parties du code d'ArmoniK, nous devons trouver une manière de créer des containers à partir de celles-ci, et de gérer ces derniers. Il existe une solution à ce problème : Kubernetes.

### Pourquoi Kubernetes ?

Kubernetes est un orchestrateur de conteneurs. C'est-à-dire qu'il va utiliser les images ou les conteneurs qui lui sont fournis et gérer leur cycle de vie. Il simplifie énormément la quantité de travail manuel à fournir pour déployer un ou plusieurs conteneurs, et représente donc un gain de temps non négligeable pour n'importe quel développeur. Pour une utilisation similaire des containers, on peut d'abord penser à la solution du docker-compose. En effet, nous utilisons Docker, et celui-ci implémente dans son engin cette fonctionnalité permettant de déployer avec des caractéristiques très précises plusieurs conteneurs, tout cela dans un simple fichier YAML, qui est très facile à lire par l'humain.

Cependant, docker-compose possède des limitations : il n'est tout simplement pas adapté pour déployer de nombreux containers sur plusieurs hôtes. Il est très pratique pour tout développement et déploiement local, mais est limité pour des projets cloud nécessitant plusieurs ordinateurs et serveurs. Kubernetes, au contraire, est une solution qui permet de déployer de nombreux conteneurs sur différentes machines. Il existe des solutions similaires, notamment proposée par Docker lui-même avec Docker Swarm.

C'est cependant Kubernetes qui est le leader en termes d'orchestration de conteneurs d'applications, avec plus de 80% d'adoptions dans les entreprises. Kubernetes a résolu de nombreux problèmes d'administration, notamment au niveau de la mise à l'échelle automatique. En informatique, il s'agit du phénomène d'augmentation de la charge et du volume nécessaire au bon fonctionnement d'une application dû à son succès. Kubernetes permet de faire du auto-scaling, et donc d'adapter les ressources des différentes parties de l'application en fonction de la demande, ce qui le rend extrêmement utile.

### Précisions sur Kubernetes

Kubernetes est donc un orchestrateur de conteneurs. Commençons en définissant la plus grande instance créée par Kubernetes lors de son déploiement : le *cluster*. Il représente une ou plusieurs *nodes*, qui sont des machines physiques ou virtuelles possédant des ressources limitées. Kubernetes fonctionne en plaçant les applications conteneurisées dans des *pods*. Celui-ci représente généralement un ensemble d'applications très dépendantes les unes des autres (comme un site web, une API et une base de données). Il peut donc être composé de plusieurs conteneurs. Les applications placées dans les pods vont garder une adresse IP<sup>5</sup> commune, toujours dans l'idée de les regrouper le plus possible entre eux. Et les *pods* sont partagés entre les *nodes* du *cluster*.

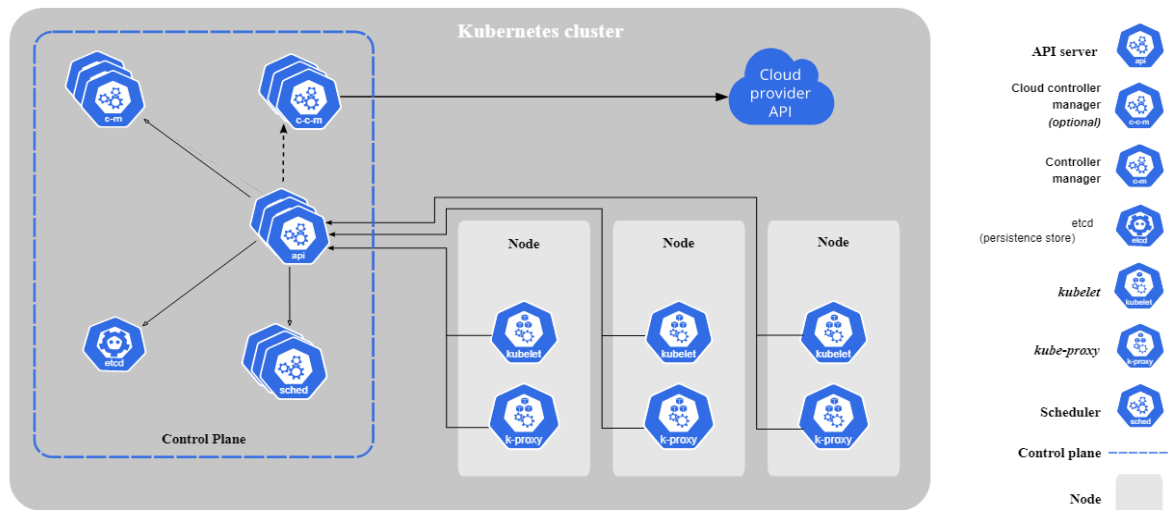


Figure 10 - Schéma de l'infrastructure d'un cluster Kubernetes

Pour partager les ressources efficacement, Kubernetes utilise un *Control Plan* ou Plan de Contrôle en français. On peut le décomposer en plusieurs sous-applications. Nous avons le *Scheduler* qui va indiquer aux *pods* non distribués vers quelle *Node* ils doivent se rendre. Pour se faire, le *scheduler* va prendre en compte les contraintes imposées par les opérateurs, les ressources disponibles et encore d'autres facteurs de diverse importance. Il est composé d'une API qui permet aux développeurs de garder un certain contrôle sur la disposition des *pods* et des ressources utilisées. Lorsque de nombreuses demandes sont faites sur l'API, la scalabilité horizontale de Kubernetes va lui permettre de déployer plus d'instance de ces serveurs.

### C. Terraform

Nous avons maintenant des images et nous sommes capables de gérer leur cycle de vie dans une infrastructure complexe fournie par Kubernetes. Mais mettre en place une infrastructure de cette taille est chronophage, et devient alors le travail des clients. Pour éviter ceci, l'équipe ArmoniK a décidé d'utiliser le logiciel Terraform.

### Pourquoi Terraform ?

Terraform est un service permettant de créer facilement une infrastructure d'application. Elle représente le squelette de l'application ArmoniK et permet de déployer et de gérer sur tout environnement la manière de laquelle les différents services d'ArmoniK fonctionnent et communiquent. Il existe évidemment d'autres outils de management d'infrastructure, dont Ansible, qui est la technologie complémentaire à Terraform. Ceux-ci ne sont pas en concurrence, elles sont adaptées à différents besoins, comme nous allons le voir.

Ansible permet de gérer plus facilement les changements de version et de configuration, de par sa capacité à décrire précisément les étapes qui doivent se succéder pour mettre en place une infrastructure. Il définit ce que l'on appelle une infrastructure mutable, qui n'est pas redéployée à chaque changement de version.

Terraform est quant à lui adapté aux changements d'infrastructure. Si bien que lorsque l'on change la version d'une application, on se doit de la redéployer entièrement. C'est ce que l'on appelle une infrastructure immuable. A première vue, elle n'a d'intérêt que pour les changements d'infrastructure, mais l'immutabilité permet entre autres de maintenir un risque d'échec de l'application à un niveau très bas.

Le problème de la mutabilité est que si l'on veut remettre en production une version précédente de l'application, un certain nombre d'étapes est nécessaire, et par conséquent plus de chance de faire une erreur coûteuse. Redéployer l'infrastructure peut paraître comme une approche plus terre à terre, mais beaucoup plus sûre dans le cadre d'une utilisation du logiciel sur de nombreux serveurs. Si l'un de ceux-ci venait à avoir un problème, cela pourrait avoir des résultats dramatiques.

En d'autres termes, Terraform simplifie énormément la mise en place d'une infrastructure, tandis qu'Ansible permet de faire facilement des modifications dedans. Il est de ce fait possible d'utiliser à la fois Terraform et Ansible pour mettre en place et gérer une infrastructure.

Ce n'est cependant pas ce qui est fait avec ArmoniK. Comme nous le verrons avec l'utilisation complémentaire de Kubernetes, ce projet n'a pas besoin d'Ansible. De plus, l'application est faite pour être déployée sur Amazon Web Services (AWS). Avec les service de cloud, en règle générale, les applications ne sont déployées que pour une quantité limitée de temps, et pour une bonne raison. Pour un outil comme ArmoniK, AWS devient un service très onéreux, si bien que les entreprises qui laissent tourner trop longtemps leurs applications peuvent perdre des milliers d'euros.

Les applications déployées sur AWS sont donc généralement détruites aussitôt leur utilisation achevée. Pour ArmoniK, qui effectue de très nombreux calculs quand ses clients en ont besoin, l'application est donc déployée temporairement sur AWS. Même si les calculs peuvent avoisiner la dizaine d'heure, l'application est détruite une fois que les résultats sont récupérés. Il n'y a donc pas d'intérêt véritable à utiliser Ansible dans le cadre d'ArmoniK, si l'application est détruite plusieurs fois entre deux mises à jour. Terraform est amplement suffisant pour une utilisation sur AWS.

### Précisions sur Terraform

Terraform se définit comme une technologie « d'infrastructure as code ». Là où généralement l'infrastructure est mise en place à la main, ici tout va être géré dans des fichiers HCL (pour Hashicorp Configuration Language). Ces fichiers fonctionnent en définissant des *providers*, qui sont les services dans lesquels pourront être injectés l'application. Nous pouvons les retrouver sur le site de Terraform, où sont évidemment référencés les *Cloud Providers* comme AWS ou Azure, mais aussi des services de type base de données, ou encore services web.

Terraform définit aussi les ressources nécessaires et maximales pour chacun des services de l'application. Il est capable de récupérer et fournir les variables d'environnement nécessaire au bon fonctionnement de toutes les sous-parties de l'infrastructure qu'il doit créer. Le HCL est un langage humainement lisible, et en devient donc rapidement compréhensible, même sans avoir de réelles compétences dans celui-ci.

Un des plus grands avantages de Terraform est sa capacité à versionner facilement les différents changements de l'infrastructure. Le versionnage est compliqué avec d'autres outils de gestion d'infrastructure, mais est absolument nécessaire pour comprendre de quelles modifications peuvent venir un problème. De plus, cela permet de voir les conséquences des changements que l'on apporte à l'infrastructure avant de les accepter. Le seul sacrifice à cet avantage est qu'il nécessite de stocker les informations sur les versions dans un fichier *terraform.tfstate*, qui dépendamment de la complexité de l'infrastructure, nécessitera une quantité de stockage utilisée plus ou moins importante.

#### D. Fonctionnement commun de K8s avec Terraform

Maintenant que nous savons pourquoi nous utilisons ces technologies et comment ces dernières fonctionnent, nous pouvons nous pencher sur leur utilisation commune. Le but de Terraform va être de simplement instancier le cluster Kubernetes et les différents conteneurs Docker qu'il va devoir gérer. Pour ce faire, nous allons déclarer ces derniers dans les fichiers HCL et préciser à Kubernetes les spécifications supplémentaires dont nous avons besoin. Une fois Terraform utilisé, nous n'avons plus qu'à nous pencher sur l'API du *Control Plan* de Kubernetes pour gérer notre cluster.

Dans le semestre précédant mon entrée en tant que stagiaire chez Aneo, j'ai pu découvrir l'utilité des conteneurs. Je n'avais alors utilisé que docker-compose pour gérer plusieurs d'entre eux. J'ai donc découvert Kubernetes et Terraform lors de mon stage, qui sont des technologies très intéressantes qui me donnent envie de les réutiliser plus tard. Pour le moment, je suis plus que novice dans ces technologies, mais j'ai plus que l'envie d'approfondir mes connaissances en ce sujet.

### 6. Des outils de taille

Maintenant que nous savons comment est organisé ArmoniK, et comment celui-ci est déployé, je vais maintenant décrire les différentes technologies sur lesquelles j'ai pu travailler ou communiquer.

#### A. Développement Frontend

La première mission de mon stage était de faire évoluer l'interface Administrateur de ArmoniK, que l'on peut décrire comme une application Web permettant d'interagir avec les différentes parties du logiciel. Pour ce faire, quelques technologies que je m'appête à expliciter ont été utilisées.

##### **Angular**

L'interface Administrateur d'ArmoniK est développée à l'aide d'un Framework Javascript appelé Angular. Elle a pour but de faciliter la gestion et la visualisation des différents services et résultats d'ArmoniK. L'utilisation d'un Framework javascript se justifie par le fait qu'elle permet d'accomplir de nombreuses choses difficilement implémentables en développement web basique. Pour ce dernier, on créait un ou plusieurs fichiers html tous reliés entre eux, qui appellent chacun un fichier Javascript. La manière de laquelle ceux-ci étaient structurés est-elle qu'ils devenaient rapidement illisibles et peu réutilisables.

Cependant, un Framework javascript se base sur les fichiers javascript plutôt que HTML lors du développement. De cette manière, il est beaucoup plus facile de structurer le code javascript. Les fichiers HTML sont gérés par ces derniers, et n'ont plus besoin d'importer les scripts, simplifiant grandement leur écriture. De plus, un des grands avantages des Framework javascript, c'est la décomposition du code en divers composants réutilisables absolument partout.

Par exemple, nous pouvons créer un composant « tableau » que nous voulons utiliser dans diverses pages. Nous pouvons déjà penser à de nombreuses choses : quelles sont les données qui lui sont envoyées, quels sont les noms des colonnes qui vont le composer ? Nous pouvons préparer le composant « tableau » à accueillir une liste de colonnes, qu'il créera et dans lesquelles il enverra les données qu'il aura lui-même reçues d'un autre composant. Ce tableau sera ensuite implémenté dans les diverses pages qui nécessitent sa présence, et pourra être encore utilisé à l'avenir, en cas de besoin.

Maintenant, il existe de nombreux Framework Javascript, les principaux étant Angular, React et Vue (dans l'ordre de création chronologique). Chacun possède ses forces et ses faiblesses, mais sont toujours très efficaces dans ce qu'ils font. La plupart du temps, ce sont les préférences des développeurs qui se font ressentir dans le choix du Framework à utiliser. Cependant, pour des projets professionnels et industriels comme ArmoniK, le choix n'est pas fait à la légère. Chaque Framework a son niveau de complexité, ce qui peut devenir une dette technique lorsque les équipes se renouvellent. Certains sont adaptés à une technologie tandis que d'autres non.

Angular possède plusieurs qualités, la plus évoquée est sans doute le fait qu'il n'est pas basé sur Javascript, réputé pour son comportement contre-intuitif, mais sur TypeScript, un cousin de ce dernier. TypeScript est un langage typé : c'est-à-dire que chaque variable du code ne peut pas être tout à la fois. On définit à l'avance le type d'une variable, ce qui simplifie énormément le développement, et donc sa maintenabilité.

De plus, Angular est développé par Google, ce qui lui donne un avantage particulier par rapport aux deux autres Framework. En effet, étant donné que les protocoles Buffer utilisés pour le protocole gRPC sont aussi développés par Google, Angular s'est vu rapidement adapté à cette nouvelle technologie et est donc le Framework le plus adapté pour ArmoniK.

Cependant, de futurs changements au niveau de l'Api d'ArmoniK, développée en C#, pourraient bouleverser ce choix. Avec l'arrivée de la version .Net 7 de celle-ci, l'intégration du gRPC sera beaucoup plus simple pour du javascript, si bien que chaque Framework pourra aisément utiliser le gRPC.

## Observables

Angular fut pour moi l'occasion de découvrir un outil informatique très pratique dans la gestion d'applications basées sur des événements, le *patron Observateur* ou *Observable pattern* en anglais. C'est un outil que j'ai pu énormément utiliser dans le cadre de mon travail.

Le *patron Observateur* est un outil informatique souvent implémenté dans les applications utilisant des événements, c'est-à-dire lorsqu'un utilisateur interagit avec elle, ou alors lorsque celle-ci reçoit de nouvelles données de manière passive. Il est par conséquent très utile sur les applications web, qui sont basées sur de tels événements.

Dans son implémentation la plus basique, le patron utilise principalement deux objets : l'*Observateur* et le *Sujet*. Le *Sujet* représente une valeur future, qui, lorsqu'elle sera reçue, sera traitée par les *Observateurs* ayant souscrit au *Sujet*, selon une méthode qui aura été définie par le développeur. Certaines implémentations, comme *RxJS* (bibliothèque javascript implémentant le patron Observateur), *décrivent* aussi un objet *Observable*, qui est très similaire à un *Sujet*. L'*Observable* se différencie alors par le fait qu'un unique *Observateur* puisse y souscrire.

Avoir des valeurs futures simplifie le travail des développeurs, car ils n'ont pas besoin de créer une variable temporaire temps qu'ils n'ont pas reçu la réponse. Une seule variable suffit, et est gérée efficacement par les langages implémentant le patron observable. Le développeur n'a qu'à préciser ce que fera la variable quand elle aura enfin reçu cette valeur future.

## Clarity

En plus d'un Framework Javascript, l'interface Administrateur d'ArmoniK utilise aussi un Framework HTML/CSS. Le principe de ces Framework est très simple : ils se basent sur un Framework Javascript et créent une grande quantité de composants et de visuels.

Développer et tester le fonctionnement et le rendu visuel de composants peut-être extrêmement chronophage et fatigant pour n'importe quel développeur. Les Framework CSS sont donc nécessaires pour ne pas perdre de temps sur des détails visuels. De plus, nous avons maintenant une grande quantité de Framework CSS disponibles pour les Framework Javascript les plus connus.

Utiliser de tels outils est donc très avantageux, d'autant plus que ceux-ci sont couramment mis à jour, et que chaque bug peut être notifié sur GitHub ou GitLab aux développeurs travaillant sur ceux-ci.

J'ai noté un peu plus tôt qu'il existait de nombreux Framework CSS pour Angular, Vue et React. On en retrouve des communs à tous, tels que Bootstrap ou Tailwind. Pour se recentrer sur Angular, nous avons chaque année, voire chaque mois, un classement des 15 meilleurs Framework CSS adapté à ce dernier. Là encore, leur utilisation dépend des besoins et des habitudes des développeurs. Certains sont pratiques pour développer des sites purement visuels tels que des portfolio ou des sites d'entreprise.

Certains, comme Clarity Design, sont plus adaptés au traitement de données, les rendant très utiles pour des projets d'interface administrateur comme celle d'ArmoniK. C'est d'ailleurs celle-ci qui a été retenue, notamment pour son composant *Datagrid*, permettant de créer, comme dans mon exemple plus tôt, des tableaux de données triables et filtrables très facilement.

Son éditeur, VMware, très connu dans le milieu de l'informatique, est une référence de confiance, contrairement à d'autres boîtes moins connues. C'est donc un autre avantage à utiliser Clarity étant donné que l'on peut s'attendre à avoir des mises à jour fréquentes ainsi que de nouvelles fonctionnalités ajoutées fréquemment.

Nous avons donc Angular et son avantage d'utilisation des protocoles google avec la facilité de traitement des données qu'offre Clarity Design qui nous permettent de créer une interface administrateur claire et concise pour ArmoniK.



## B. gRPC

L'interface administrateur doit nécessairement accéder aux données du logiciel pour pouvoir avoir la moindre utilité. Pour se faire, ArmoniK a implémenté une API qui sert justement à partager des données et à agir sur le fonctionnement de certaines parties du logiciel. Pour se faire, ArmoniK a utilisé une technologie très récente, le gRPC, pour google RPC.

Cette technologie est basée sur le RPC et le HTTP/2. Le RPC (Remote Procedure Call) est un protocole réseau qui permet à un programme, le client, de communiquer avec un serveur de manière simple, sans avoir de détail sur son réseau. C'est un protocole synchrone, c'est-à-dire que le programme client va attendre de recevoir une réponse avant de continuer son exécution. Le HTTP/2 est l'évolution du très connu protocole HTTP que nous pouvons retrouver dans n'importe quelle adresse de site web. La première version de celui-ci, créé en 1990, permettait d'accéder au *world wide web* aka *www*. Sa deuxième version, créée en 2014, est une évolution majeure, dans le sens où il nécessite de réaliser moins de requêtes auprès des serveurs pour récupérer les données d'une page web. De ce fait, le temps de chargement d'une page ou application web est grandement réduit et optimisé.

Le gRPC est un framework RPC qui permet de créer facilement une API très légère en métadonnées. Cela passe par la mise en place de plusieurs outils, qui sont intégrés à l'aide des fichiers *protobufs*. Ceux-ci ont une syntaxe très simple qui permet de définir des objets et des méthodes capables de les retourner. Ces fichiers sont partagés entre le serveur gRPC et les programmes clients. Les serveurs gRPC vont implémenter ces méthodes tandis que les clients vont créer des *stubs* gRPC. Ces *stubs* permettent aux programmes clients d'avoir les mêmes méthodes que le serveur gRPC. Les *stubs* transmettent la requête qu'ils envoient au serveur gRPC aux clients pour qu'ils le gèrent comme un objet local.

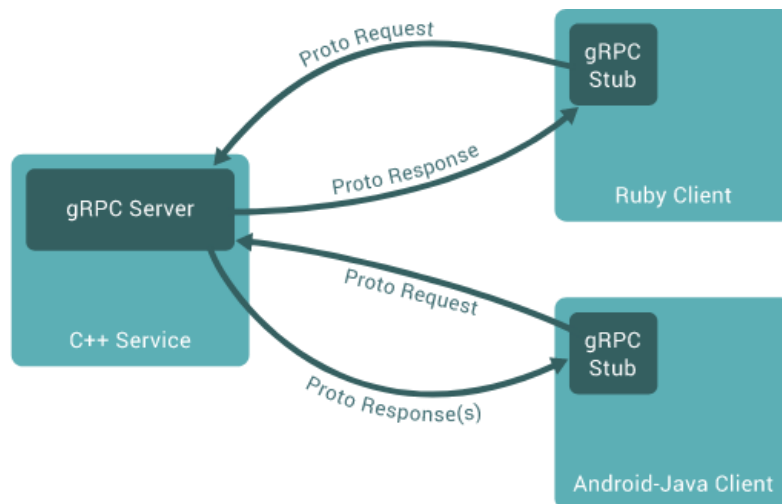


Figure 11 - Schéma de communication entre serveur gRPC et deux applications clientes

Le gRPC apporte plusieurs choses :

- Il est environ 10 fois plus rapide qu'une API basique basée sur le REST et JSON.
- Il permet d'envoyer ou de recevoir des flux continus de données en une seule requête, ce qui réduit la quantité de données nécessaire pour recevoir ou donner plusieurs objets différents.



Un des avantages à utiliser le gRPC avec un client codé en *Javascript* ou *TypeScript*, c'est que les réponses sont transmises sous la forme d'*Observables*. De ce fait, nous pouvons gérer les objets retournés par les appels gRPC comme des objets futurs, avec tous les avantages que cela apporte.

### C. Nginx

Maintenant, nous avons tous les outils pour créer une application web capable de gérer des appels à un serveur. Mais encore faut-il être capable d'accéder à cette application depuis ArmoniK. Pour se faire, il faut servir l'application, ou, en d'autres termes, la placer sur un serveur Web.

Pour se faire, nous allons utiliser Nginx. Il s'agit d'un logiciel de serveur Web, qui peut aussi être utilisé en tant que proxy inverse. Il permet notamment d'héberger une application web, en gérant le chemin qui lui est passé via une URL<sup>6</sup>. En tant que proxy inverse, il permet de renvoyer un utilisateur sur divers serveurs web, toujours en prenant en compte les arguments qui lui sont passés via une URL.

## 7. Implication dans le projet

### A. Seq et Grafana

ArmoniK est une application faisant tourner de nombreux calculs et tâches. De ce fait, ces tâches peuvent subir des erreurs. De même pour certaines fonctionnalités d'ArmoniK. De ce fait, il est nécessaire d'avoir des applications de monitoring<sup>7</sup> pour ces erreurs. ArmoniK a choisi d'utiliser Seq pour les erreurs internes du logiciel et Grafana pour monitorer les tâches en cours.

Ce sont eux aussi des applications web déployées avec ArmoniK. De ce fait, elles peuvent ne pas être disponibles dès le départ de l'application. Hors, l'interface Administrateur est censée permettre d'accéder à ces espaces de monitoring. De ce fait, cette dernière doit pouvoir détecter l'état des applications de monitoring, et agir en conséquence. Si une des applications n'est pas disponible, aucune interaction avec elle ne sera proposée à l'utilisateur de la GUI.

Afin de détecter la présence de ces applications sur l'interface, j'ai dû utiliser un système d'observable. Grâce à ceux-ci, j'ai pu détecter leur état, et en fonction de celui-ci, j'ai pu empêcher l'interaction de l'utilisateur avec eux.

### B. Filtres

Les informations que transmet ArmoniK sont pour la plupart affichées dans des tableaux. Ces tableaux sont des interfaces fournies par le framework Clarity, que j'ai pu présenter plus tôt. Ce dernier permet d'ajouter des filtres à chaque colonne du tableau. Certains filtres sont déjà disponibles pour différents types de colonnes, mais malheureusement ne suffisent pas pour les données d'ArmoniK. De ce fait, j'ai dû créer plusieurs filtres personnalisés à l'application et les ajouter aux colonnes de ces tableaux.

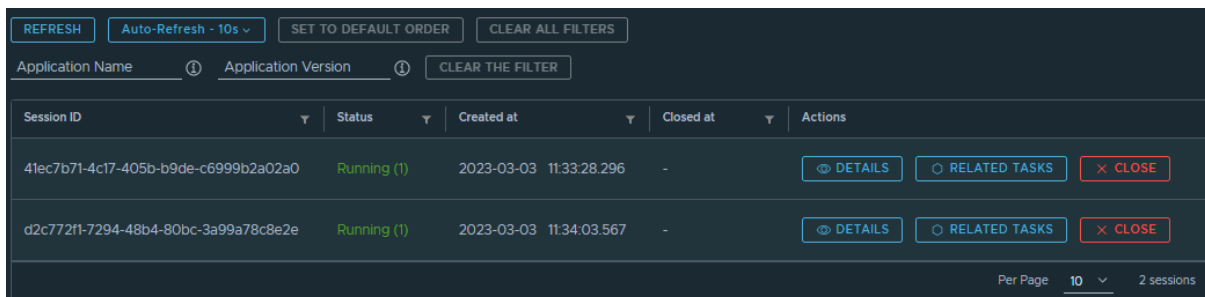
Pour définir le fonctionnement de ces filtres, je me suis inspiré des fichiers protos définis par les API. Ceux-ci implémentent déjà les appels de récupération de données pour chaque

tableau, en comprenant les filtres. De ce fait, je savais quels types de données les filtres devaient sortir pour pouvoir être acceptés par les API. En étudiant ces différents filtres, j'ai remarqué des similitudes entre certaines colonnes des différents tableaux. Ces colonnes avaient des données ayant une forme similaire, mais acceptant des valeurs différentes. Par conséquent, je devais implémenter des filtres qui soient le plus génériques possible. C'est-à-dire que ces filtres soient capables de traiter des données de ces différentes colonnes.

J'ai donc eu l'occasion de créer un total de cinq composants de type filtre pour l'ensemble des tableaux d'ArmoniK :

- Select Filter : Il s'agit d'un filtre permettant de sélectionner une valeur dans une liste affichée et l'appliquer à la colonne. La liste doit être fournie au composant pour être affichée, ce qui signifie qu'il peut être réutilisé pour des listes différentes.
- Combobox Filter : De la même manière que le précédent, il affiche une liste de valeurs étant applicables en tant que filtre. La différence réside dans le fait que plusieurs valeurs sont sélectionnables en même temps. Ce filtre est préféré au précédent lorsque la liste des valeurs est très grande.
- Id Filter : Il permet de filtrer simplement les champs renseignés par la valeur donnée par l'utilisateur. Il est généralement utilisé pour l'identifiant des résultats, d'où son nom « id ».
- Numeric filter : Ce composant permet de la même manière que le précédent, de filtrer le champ renseigné par la valeur donnée au filtre. Cependant, il n'accepte que des chiffres et des numéros.
- Date filter : C'est un composant capable de filtrer les résultats de l'appel GRPC en fonction de leur date supérieure et inférieure. Il possède donc deux champs pour sélectionner ou renseigner la ou les dates souhaitées.

Certains filtres peuvent être appliqués même s'ils ne sont pas reliés à une colonne particulière. J'ai alors dû trouver une manière différente de les gérer, sans pour autant nuire à la lisibilité de l'application web. Evidemment, il a fallu implémenter une manière de les supprimer intuitivement. Deux manières de le faire ont été mises en place. La première implémente un bouton « supprimer le filtre » dans l'interface de chaque filtre. La deuxième est l'ajout d'un bouton « supprimer tous les filtres » au-dessus des différents tableaux, sur les pages utilisant les filtres.



The screenshot shows a web interface for managing sessions. At the top, there are buttons for 'REFRESH', 'Auto-Refresh - 10s', 'SET TO DEFAULT ORDER', and 'CLEAR ALL FILTERS'. Below these are input fields for 'Application Name' and 'Application Version', with a 'CLEAR THE FILTER' button. The main table has columns: Session ID, Status, Created at, Closed at, and Actions. Two rows of sessions are visible, both with status 'Running (1)'. Each row has buttons for 'DETAILS', 'RELATED TASKS', and 'CLOSE'. At the bottom right, it says 'Per Page 10' and '2 sessions'.

Session ID	Status	Created at	Closed at	Actions
41ec7b71-4c17-405b-b9de-c6999b2a02a0	Running (1)	2023-03-03 11:33:28.296	-	<a href="#">DETAILS</a> <a href="#">RELATED TASKS</a> <a href="#">CLOSE</a>
d2c772f1-7294-48b4-80bc-3a99a78c8e2e	Running (1)	2023-03-03 11:34:03.567	-	<a href="#">DETAILS</a> <a href="#">RELATED TASKS</a> <a href="#">CLOSE</a>

Figure 12 - Page Session d'ArmoniK

Sur l'image ci-dessus, nous pouvons voir le tableau des sessions de calculs d'ArmoniK. Nous pouvons voir que les 4 colonnes traitant des données sont filtrables, et qu'en plus, certains filtres sont ajoutés sans qu'il n'y ait de colonnes. Nous pouvons aussi observer un bouton qui permet d'effacer l'ensemble des filtres appliqués à la page.

```
172.31.67.64:5000/admin/fr/tasks?status=3&createdBefore=1679439600000
```

Figure 13 - URL de la page tâches filtrée

Les données de ces filtres sont stockées dans l'URL de la page web. De cette manière, les différents utilisateurs souhaitant se partager les pages avec les mêmes filtres sont capables de simplement passer cette dernière.

### C. Nouvelles pages !

Dans le cadre de mon apprentissage de Angular et de la structure du code de l'interface Administrateur, j'ai dû implémenter les page « applications » et « partitions » d'ArmoniK. Pour l'orchestrateur, plusieurs types d'applications de calculs peuvent être utilisés. Ils sont alors identifiés et séparés les uns des autres. Chaque session de calcul, chaque tâche et chaque résultat est de fait lié à une application. Une application peut avoir plusieurs sessions de calculs. Les partitions peuvent être vues comme des espaces de calculs sur lesquels les applications peuvent être déployées.

L'API d'ArmoniK a implémenté les fonctions permettant de récupérer les données des applications et des partitions pendant la période du stage. J'ai donc pu alors m'inspirer des API et des autres pages déjà créées, avec les fonctions des fichiers protos pour créer cette page. Ce fut un très bon exercice, qui réconfortait mes connaissances dans le code de l'application.

### D. Gestions de différents bugs et petites fonctionnalités

Lors de l'implémentation de nouvelles fonctionnalités, il y a toujours une possibilité qu'un bug soit introduit sans pour autant avoir été remarqué plus tôt. Ce genre de bug n'est pas forcément testable, et est majoritairement dû aux outils utilisés, qui ne font pas toujours exactement ce que le développeur pense. Donc lorsqu'un nouveau problème est découvert, il faut le régler le plus vite possible. Ce fut donc une partie de mes missions, qui ne prenait cependant pas énormément de temps, et qui était due à une prise d'initiative de ma part, la plupart du temps.

### E. Reviews

L'implémentation d'une nouvelle fonctionnalité signifie une chose : qu'une *Pull Request* va être ouverte. Cela implique que quelqu'un doit relire le code de cette *PR* et demander des modifications si nécessaire.

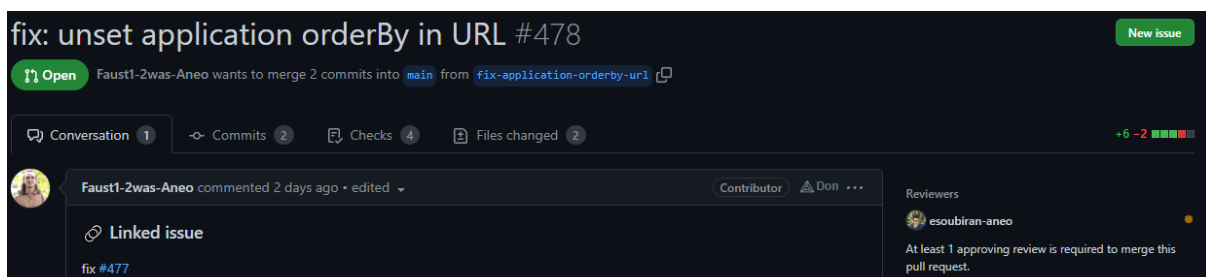


Figure 14 - Pull Request GitHub

Ce fut donc pour moi une mission relativement complexe au début du stage, étant donné que je ne connaissais ni la structure du code, ni Angular. Cependant, avec le *Trunk-Based development*, ce fut pour moi l'occasion d'apprendre sur des petits morceaux de code. En 20

semaines de stage, ce ne fut pas moins de 200 revues de codes réalisées sur ce dépôt Git partagées entre deux membres.

## F. Actions GitHub

J'ai référencé les actions GitHub, aussi connu sous le nom de pipelines, comme une méthode intéressante pour s'assurer du bon état du code tout en étant un outil très utile pour le DevOps, notamment au niveau de l'*Intégration Continue* et du *Déploiement Continu*. J'ai, par quelques occasions, eu l'occasion de travailler sur ces pipelines pour les optimiser, ou les adapter à une nouvelle partie du code.

Les machines GitHub s'occupant des pipelines sont mises à disposition gratuitement à n'importe qui. Cela signifie qu'elles sont forcément limitées en puissance et qu'elles peuvent être rapidement surchargées par de nombreux pipelines. Lorsque de nombreuses Pull Request sont ouvertes et que des modifications de code sont poussées sur chacune d'entre elles presque simultanément, les ressources sont vite saturées, et cela résulte en des problèmes. Les pipelines sont bloqués et s'arrêtent. Cet arrêt peut avoir lieu à n'importe quel moment, au début comme à la fin de chaque action.

Cela peut devenir un réel problème pour les développeurs et les DevOps, qui doivent alors relancer manuellement les pipelines et être sûrs que celles-ci finissent correctement. Cela peut même jouer sur le temps de travail. En effet, si certaines actions durent 5 minutes, d'autres peuvent durer 10 voire 20 minutes. Cela devient donc extrêmement chronophage de devoir redémarrer les pipelines à cause d'un souci de puissance, d'autant plus si la fonctionnalité qui lui est associée est nécessaire pour avancer sur une autre.

Il est donc important d'optimiser au mieux ses pipelines pour éviter ce genre de problèmes. Nous nous en sommes rendu compte courant décembre, lorsque nous nous sommes effectivement confrontés à ce problème. Celui-ci avait lieu à cause de la génération des fichiers enfants des « protoc », ayant lieu pendant la majorité des étapes. Leur génération si fréquente mobilise les ressources de la machine GitHub, et une des pipelines s'arrête alors pour laisser une autre continuer.

La solution a alors été d'utiliser un système de « cache », qui consiste à générer une unique fois les fichiers enfants des « protoc » et les mettre à disposition dans un dossier accessible par toute la machine GitHub. Étant donné qu'ils sont stockés dans la machine lors qu'une action les crée, une autre action pourra alors aller les rechercher à l'endroit indiqué plutôt que de les régénérer une énième fois. Et si les fichiers protoc sont modifiés, alors la machine le remarquera lors d'un pipeline et les régénère. Le développeur n'a alors besoin de faire attention aux caches présents dans la machine GitHub, en les supprimant de temps en temps, afin qu'elle ne soit pas pleine. Et pour y arriver, pas d'autres choix que d'y aller à la main !

## G. Nginx (Ingress)

Au milieu de mon stage, nous avons repéré un problème sur le système de traduction utilisé par l'interface administrateur. Ce système implémente une traduction à la volée. Cela signifie que le texte traduisible de l'interface est stocké dans différents fichiers, un fichier à une ligne. L'application web va laisser l'utilisateur choisir la langue qu'il souhaite utiliser, et rechercher la traduction appropriée dans le fichier correspondant à la langue sélectionnée. C'est un système qui fonctionne efficacement, mais qui représentait un problème majeur : il

s'agissait d'un module supplémentaire à l'application qui n'était pas mis à jour depuis quelques mois.

Ce qui veut dire que les différents bugs et problèmes découverts ne seraient pas corrigés. De ce fait, nous avons décidé d'utiliser un autre système de traduction, totalement adapté à Angular, étant donné qu'il était développé pour ce Framework. Plutôt que d'utiliser une traduction à la volée, plusieurs versions de l'application sont construites, une version correspondant à une langue. Le travail du développeur devient cependant un peu plus complexe : il doit pouvoir servir les différentes langues de l'application.

Ma mission fut donc de réaliser un tel service, à l'aide de Nginx. Il a fallu que je modifie la configuration du fichier Nginx présent dans le déploiement Terraform pour que l'utilisateur, en fonction de la langue de préférence de son navigateur, soit redirigé vers une des versions de l'application web.



Figure 15 - Avant modifications

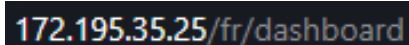


Figure 16 - Après modifications

Nous pouvons donc voir que deux versions sont servies par le serveur, une anglaise et une française. La version d'abord choisie en fonction de la langue du navigateur, sachant que si la langue n'est pas disponible, la valeur de base est l'anglais. Toutes les versions restent cependant disponibles pour l'utilisateur, ce qui signifie qu'il peut passer d'une langue à l'autre en modifiant le « /fr/ » de l'URL en « /en/ ».

#### H. Terraform (Double service)

Comme j'ai pu le mentionner plus tôt, l'interface administrateur a reçu beaucoup de modifications pendant mon stage. Son fonctionnement a changé au plus profond, modifiant la manière dont les pages sont gérées. L'ancienne API a été supprimée, pour être remplacée par une API gRPC. En bref, énormément de changements. Ceux-ci ont évidemment un impact sur les utilisateurs de l'application. Le chef d'équipe d'ArmoniK, en voyant les changements apportés à l'interface administrateur, a trouvé qu'ils étaient trop importants pour les intégrer du jour au lendemain sur ArmoniK.

La décision a alors été prise de déployer à la fois la nouvelle et l'ancienne version de l'interface administrateur. Cela passe par plusieurs étapes : modifications du nginx pour ajouter un service sur l'ancienne version, et le déploiement d'un nouveau conteneur pour cette dernière.

J'ai donc été choisi pour réaliser cette tâche. Celle-ci s'est passée majoritairement par la modification de la configuration Terraform du projet. C'était pour moi quelque chose de très intéressant, car cela me faisait sortir du développement web pour m'occuper de l'infrastructure d'une application. De plus, je n'avais jamais fait de Terraform avant, et bien que je ne puisse pas encore dire que je connaisse exactement la structure d'ArmoniK, j'ai compris des choses sur le fonctionnement de cette technologie.

## 8. Difficultés rencontrées

### A. Commit Git

Comme j'ai pu l'expliquer plus tôt, il est important d'avoir un historique Git peu pollué. Cela revient à faire attention au nombre de commits réalisés dans une *PR*, ou à *squasher* ces derniers pour n'en faire qu'un seul. C'est avec cette méthode que nous travaillons sur le dépôt de l'interface administrateur. Cependant, sur les autres dépôts, les *PR* ne sont pas forcément *squashées*. Ce qui signifie qu'il faut faire attention à avoir un nombre minimal de commits. C'est à ce niveau là que j'ai eu beaucoup de difficultés, notamment sur les technologies que je ne maîtrisais pas. A force de corriger mes erreurs et mes oublis sur celles-ci, mon nombre de commits augmentait vite. J'ai bien évidemment commencé à m'améliorer sur ce point en comprenant davantage la structure du code, mais le mal était déjà fait dans la plupart des *PR*.

### B. Nginx

Mon travail sur nginx fut la partie la plus simple à expliquer, mais malheureusement la plus difficile à réaliser. Nginx est un logiciel possédant une syntaxe relativement simple. Cependant, le service d'applications sur le web possède de nombreuses nuances que je ne connaissais pas. De ce fait, j'ai passé beaucoup de temps à essayer d'appliquer des solutions qui, d'une manière ou d'une autre, ne marcheraient pas. Il m'a fallu énormément d'aide pour comprendre les quelques nuances et résoudre mes problèmes.

### C. Pipelines GitHub

Je n'ai pas spécialement eu de difficultés à modifier les pipelines GitHub. C'est un processus un peu long, car chacune d'entre elles sur le dépôt de l'interface administrateur dure environ 5 minutes. Cependant, les machines GitHub, limitées en puissance, ont parfois des comportements très étranges. De ce fait, des pipelines fonctionnels peuvent ne plus l'être à n'importe quel moment, ce qui représente une perte de temps importante. Fort heureusement, à la fin du stage, après de nouvelles modifications, ce problème n'est plus rencontré.

### D. Reproduction de problème

J'ai parfois rencontré des bugs dans mon utilisation d'ArmoniK. Ils étaient généralement dus à certaines modifications, et surtout au fait que je l'utilisais de manière locale, sur un pc portable. Bien évidemment, il y avait dans la plupart des cas l'interface chaise-clavier, qui n'avait pas connaissance du fonctionnement exact de l'application.

Dans les cas où je rencontrais un de ces bugs, je devais créer une issue sur GitHub pour le signaler, en donnant une explication exacte des étapes à reproduire pour y arriver. Bien que j'étais au courant de tout cela, je n'ai pas réussi à fournir suffisamment d'explications du premier coup. Je n'avais qu'une utilisation simple d'ArmoniK, comparé aux autres développeurs, et de ce fait n'arrivait pas à leur expliquer correctement mes problèmes.

### III- ArmoniK de couleur taillé dans le bois

J'ai donc pu travailler pendant 20 semaines sur une face importante du logiciel ArmoniK. En plus de faire grandement évoluer la partie du projet sur laquelle j'ai travaillé, j'ai eu l'occasion de découvrir de nombreuses technologies et méthodologies. En discutant avec des étudiants en M2 à l'Efrei, j'ai découvert que certaines d'entre elles allaient être enseignées au prochain semestre. Il est alors évident que mon intérêt pour ces découvertes s'en est trouvé renforcé, avec l'envie d'apprendre qui était déjà présente. Plus globalement, ce stage fut pour moi l'occasion d'approfondir mes connaissances dans certains domaines, mais surtout de les utiliser dans un contexte professionnel et de les faire évoluer.

Mais avant de commencer à discuter sur ce que j'ai appris, j'aimerais faire un point sur l'entreprise qui m'a accueillie, Aneo. Je ne la connaissais pas du tout avant de commencer la recherche de stage, et j'ai été très agréablement surpris en arrivant sur place. L'entreprise n'est pas gigantesque, elle garde une taille et une organisation très humaines. Découvrir une telle chose m'a beaucoup plu, car lors de cours de communication à l'Efrei, j'ai pu étudier quelques pratiques mises en place par les entreprises pour convaincre leurs collaborateurs de rester. Pendant ces cours, nous nous intéressions plutôt à de grandes et parfois anciennes entreprises. Ce fut donc pour moi l'occasion de voir ce qui se mettait en place dans une moyenne entreprise, dans laquelle les rapports humains sont très simples.

Le mode de fonctionnement en hiérarchie plate représente un vrai plus pour l'entreprise, j'en suis persuadé. Étant donné qu'il a été mis en place tôt chez Aneo, cela a grandement simplifié l'intégration d'un tel processus. Si bien qu'aujourd'hui les processus administratifs sont rapides et faciles, et les relations entre les différents services de l'entreprise sont bien plus que correctes. Le Flex-office est aussi une découverte intéressante, qui m'a permis de m'intégrer avec des personnes hors de mon domaine de travail. Enfin, j'ai été très surpris de retrouver des événements dans une entreprise. Avec notamment les *Aneo lunch* et les après-midi *Knowledge*, qui m'ont positivement étonné. Je n'avais jamais entendu l'idée de proposer des cours entre collaborateurs avant de rejoindre Aneo, et même si je n'ai pas eu l'occasion d'en faire autant que prévu, j'aimerais vraiment redécouvrir ce rituel dans d'autres expériences professionnelles.

J'ai aussi pu redécouvrir un outil que je pensais connaître, Git. Outil que j'ai appris à utiliser en L2, et que j'utilise depuis dans n'importe quel projet, je pensais maîtriser suffisamment l'outil pour un projet industriel comme ArmoniK. J'ai vite compris que je me trompais quand à mes compétences d'utilisation de git. J'avais des œillères qui limitaient mon utilisation et ma compréhension de l'outil. Cependant, grâce à mes collègues et de nombreux efforts tout au long du stage, je pense avoir fait grandement évoluer mes connaissances de l'outil. Aujourd'hui, il n'est pas qu'un simple dépôt de code, il permet, avec d'autres outils, comme GitHub et ses pipelines, d'être un compagnon presque nécessaire pour un développeur, comme pour un opérateur.

Ce stage fut donc pour moi l'occasion de découvrir concrètement ce qu'est le DevOps, de par le biais des pipelines GitHub. Cette organisation particulière d'une équipe, qu'on m'a présenté à maintes reprises pendant mon cursus, est en effet très intéressante. Je pense n'avoir que éraflé la surface de cette méthodologie de travail, mais je pense avoir déjà compris l'importance et l'efficacité de celle-ci. Grâce au DevOps, et les aspects de tests d'intégration et



de tests continus, n'importe quelle application est capable d'être toujours fonctionnelle. J'ai énormément apprécié cette approche projet, qui permet d'éviter et de gérer rapidement les erreurs de déploiement d'application. C'est une approche plus qu'intéressante pour des logiciels qui nécessitent de nombreuses années de fonctionnement, et permettent de faire évoluer les technologies de celui-ci sans risquer d'avoir des problèmes chez le ou les clients.

J'ai aussi appris à utiliser Terraform et Kubernetes, deux technologies fonctionnant efficacement ensemble, et très pratiques dans le cadre d'ArmoniK. Bien que je n'ai pas vraiment appris à utiliser Kubernetes et que je n'ai qu'effleuré la surface de Terraform. C'était aussi une bonne introduction à un sujet que je vais étudier dans le semestre suivant le stage. J'ai pu découvrir une manière de créer une infrastructure logicielle complexe avec Terraform et comment la gérer facilement avec Kubernetes. En faisant des recherches, j'ai pu approfondir un peu mes compétences en ce domaine, voir les points forts et faibles de chaque technologie. J'aimerais donc découvrir plus en profondeur Ansible, qui est plus adapté pour des logiciels non destinés au cloud, qui doivent être déployés pendant un très long moment. En résumé : cette partie d'ArmoniK, même si ce n'était pas un sujet qui me concernait à la base, m'a grandement intéressé et j'ai hâte de bâtir des compétences dans ce domaine.

Pour rester dans le thème des technologies découvertes, je pense avoir fait une découverte plus qu'intéressante avec le gRPC. Cette technologie de google permet de simplifier considérablement le processus de création d'API, via la technique de *Remote Procedure Call*. L'API générée par le gRPC est gérée comme de simples fonctions du programme, il est donc très facile de l'utiliser pour n'importe quel développeur. De plus, cette technologie utilise le protocole HTTP/2, signifiant que les appels API sont beaucoup plus légers, et par conséquent, plus rapides et plus écologiques. Je ne peux qu'imaginer retrouver cette technologie dans de plus en plus de projets futurs. En faisant mes recherches sur elles, j'ai pu d'ailleurs trouver que Netflix et NordVPN utilisaient gRPC pour réaliser les appels les plus optimisés possibles.

J'ai certes énormément appris pendant ce stage, mais j'ai aussi fourni un travail conséquent sur la partie web du projet. Bien que je n'avais pas de compétences en Angular au début de mon stage, j'ai su rapidement apprendre pour être le plus efficace possible sur les nombreuses tâches qui m'ont été données. En 20 semaines, c'est près de 200 tâches plus ou moins complexes que notre équipe de deux personnes ont pu écrire et relire. Je pense avoir fourni un travail de très bonne qualité, avec un code fonctionnel et optimisé. J'ai su remettre en cause les solutions que j'avais mis en place en fonction des avis de chacun, et accompagné de nombreux principes de programmation, le code actuel de l'application est très efficace. Je suis fier d'avoir tant fait évoluer un projet sur lequel je ne connaissais rien et qui me paraît très intéressant aujourd'hui.

De manière très symbolique, la release finale de la nouvelle version de l'application web d'ArmoniK s'est faite le dernier jour de mon stage. Je peux donc aisément confirmer que j'ai réalisé toutes les tâches que l'on me demandait à temps. A ce niveau-là, tout comme au niveau apprentissage, je suis donc heureux de pouvoir dire que ce stage est une véritable réussite. J'ai fait gagner un temps considérable à l'équipe d'ArmoniK et j'ai appris de nombreuses choses qui me seront, j'en suis sûr, très utiles à l'avenir.

En parlant d'avenir, j'ai pu commencer à découvrir de nombreux métiers dans l'informatique qui seraient susceptibles de m'intéresser. Bien que je connaissais théoriquement certains d'entre eux, le peu de pratique que j'ai eu lors de ces 5 mois m'a permis de me faire un début



d'avis sur ces métiers. Ce stage est une première expérience dans le milieu de l'informatique, et est là pour renforcer le bagage professionnel. Etant quelqu'un de très curieux, je n'ai jamais vraiment réussi à faire un choix vers un projet professionnel en particulier. Et ce stage m'a permis de savoir ce que j'ai envie de découvrir prochainement, que ce soit pour du DevOps ou plus simplement du développement logiciel. Il y a d'autres métiers que j'aimerais aussi découvrir, et ce stage m'a permis de me remotiver sur ce sujet. J'ai envie de faire plusieurs métiers dans ma vie, tous liés à l'informatique, mais avec une approche différente à chaque fois. J'ai aussi envie de travailler sur des projets aussi intéressants que celui d'ArmoniK, que je sens qu'ils vont être utiles, et pas une simple copie d'une application faisant les choses mieux.

Ce stage était là pour que j'apprenne. Et j'ai énormément appris, dans un cadre qui était plus qu'humain, et que j'espère retrouver à l'avenir. Aneo m'a énormément plu dans sa manière de gérer ses employés, j'espère retrouver une ambiance similaire dans mes futurs postes. La méthode de management de l'équipe d'ArmoniK était aussi quelque chose de bienvenu. Un développeur n'a pas besoin de pression pour fournir un travail à temps, et ce stage m'en a apporté la preuve. Cette méthode particulière est peut-être difficile à mettre en place pour la plupart des équipes, mais elle était très adaptée à ArmoniK.

Maintenant que mon stage est fini et que je retourne à l'Efrei, je suis très curieux de ce que les prochains semestres à l'Efrei vont m'apporter en bagage technique. J'ai envie de confronter mes connaissances acquises lors de ce stage aux cours que je vais avoir, et j'espère apprendre davantage sur ces technologies qui me semblent si intéressantes. Entre la fin du stage et le rendu de ce rapport, j'ai appris que j'étais accepté pour le parcours alternance en M2. Je pense que cette expérience sera aussi quelque chose de très bénéfique pour moi, que je revienne chez Aneo ou que j'aille dans une autre entreprise.

## Anecdotes

Pendant 20 semaines dans une entreprise, on peut s'attendre à ce que quelque chose d'inattendu arrive. Ce ne sont pas forcément des choses très impressionnantes, mais qui font de bonnes petites anecdotes.

### 1. Work-Flo

La première chose qu'on m'a dite quand j'ai rejoint Aneo, c'est qu'il fallait que je m'attende à rencontrer des Florian, Florent, et autres prénoms diminués en Flo. Et effectivement, il y en avait déjà quelques-uns à mon arrivée. Mais pendant toute la durée du stage, de nouveaux recrutements ont été réalisés, et d'autres Flo ont rejoint l'aventure Aneo.

### 2. Electricité statique

A peu près vers la moitié de mon stage, en plein hiver, j'ai commencé à être recouvert d'électricité statique lorsque j'étais au siège d'Aneo. N'importe quel objet présent dans les locaux était susceptible de m'électriser. Que ça soit un ordinateur, une table, une plante verte, ou l'eau versée par la fontaine, je risquais à tout moment de prendre un choc électrique. Ce n'était heureusement pas douloureux, juste surprenant.

### 3. Croissantage

Le croissantage est une technique commune dans les entreprises d'informatique. Cela consiste à se rendre sur l'ordinateur déverrouillé d'un collègue et d'écrire dans une conversation publique « je ramène des croissants demain ». C'est un petit piège qui punit les oublis de verrouillage de session. Deux semaines avant la fin de mon stage, je me suis fait croissanter une fois, alors même que je faisais attention avant le début de celui-ci. C'est une pratique qui a son avantage, notamment si l'ordinateur de la personne contient des informations confidentielles. Alors ne surtout pas oublier le raccourci pour verrouiller sa session sur Windows : Windows + L !

### 4. Classement des Associations

Le 2 décembre 2022, Aneo m'a invité, moi et d'autres stagiaires de l'Efrei ont été invité à un évènement particulier : le *classement des associations*. Il s'agit d'une organisation s'occupant de gérer un concours entre les différentes associations étudiantes de France. J'ai pu, avec mes autres collègues, assister à la finale de ce concours à l'opéra Bastille. C'est un évènement qui est suivi et subventionné par Aneo depuis de nombreuses années.

C'est quelque chose de très cocasse, étant donné que dès ma première année à l'Efrei, j'ai toujours été très impliqué dans la vie associative de l'école. Même pendant le stage, je n'ai pas abandonné les associations dans lesquelles je travaillais, et j'ai pu en rejoindre une nouvelle... J'ai trouvé cette coïncidence très agréable. Je ne connaissais pas son existence, et je n'ai jamais entendu une association de l'Efrei y faire allusion. Cependant, à la fin de mon stage, j'ai pu voir *Le Live*, l'association de musique de l'école, essayer d'y participer !

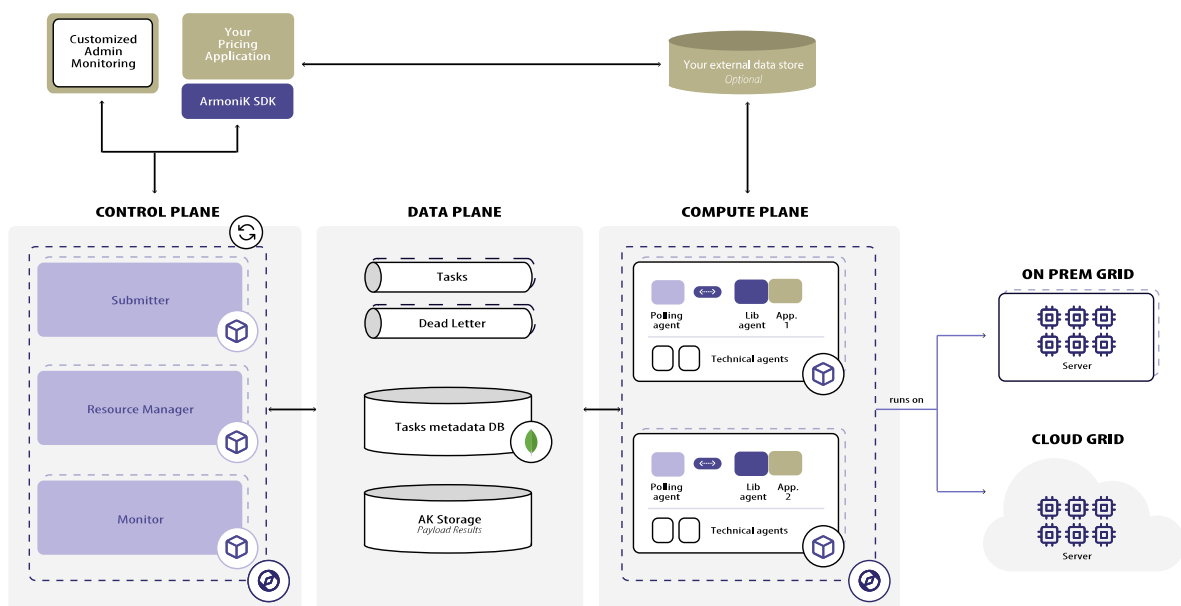
# Annexe

## Dictionnaire

1. **Application Web** : Une application Web est un logiciel utilisable dans un navigateur web. Elles permettent à un utilisateur d'interagir avec certaines parties de l'application.
2. **API (application programming interface ou interface de programmation d'application)** : Une API est une interface entre deux logiciels ou deux parties de logiciel, permettant le partage de données entre ceux-ci.
3. **Dettes Techniques** : En informatique, une dette technique représente un morceau de code vieillissant et inefficace. Elle a un impact sur la manière de coder et sur l'efficacité du logiciel. Il est donc important d'éviter d'en avoir le plus possible.
4. **Mise en production** : Lors qu'un code est fonctionnel et que les managers décident de « livrer » le code aux clients, une mise en production est faite.
5. **Adresse IP** : Il s'agit d'un numéro attribué à une machine ou un réseau connecté à Internet.
6. **URL (Uniform Resource Locator)** : Sur Internet, une URL est l'adresse d'un site. Elle peut être soit composée d'une adresse IP, soit d'un nom de domaine.
7. **Monitoring** : Le monitoring est une technique permettant de surveiller l'état d'un ou de plusieurs applications en temps réel. Cette surveillance se fait généralement par le biais de logiciels tiers, qui sont ajoutés à une application.

## Schémas

Schéma de la structure d'ArmoniK :



## Sources

Site officiel d'ArmoniK : <https://www.armonik.fr/>

Informations numériques sur Aneo : <https://www.societe.com/>

Informations sur l'orchestration informatique :

[https://fr.wikipedia.org/wiki/Orchestration\\_informatique](https://fr.wikipedia.org/wiki/Orchestration_informatique)

Enquête DORA :

<https://cloud.google.com/architecture/devops/devops-tech-trunk-based-development?hl=fr>

Informations sur la rétrospective agile :

<https://www.neatro.io/retrospective-agile/#originesetpopularisationdelartrospectiveagile>

Documentation de git : <https://git-scm.com/doc>

Informations sur le gRPC :

- <https://grpc.io/>
- <https://fr.wikipedia.org/wiki/GRPC>

Informations sur Kubernetes : <https://kubernetes.io/>

Information sur Terraform et Ansible :

- <https://www.spiceworks.com/tech/devops/articles/terraform-vs-ansible/>
- <https://www.ansible.com/blog/ansible-vs.-terraform-demystified>

Informations sur les observables :

- <https://rxjs.dev/>
- [https://en.wikipedia.org/wiki/Observer\\_pattern](https://en.wikipedia.org/wiki/Observer_pattern)

Information sur Angular : <https://angular.io/>