Projet Intégration de services et micro-services

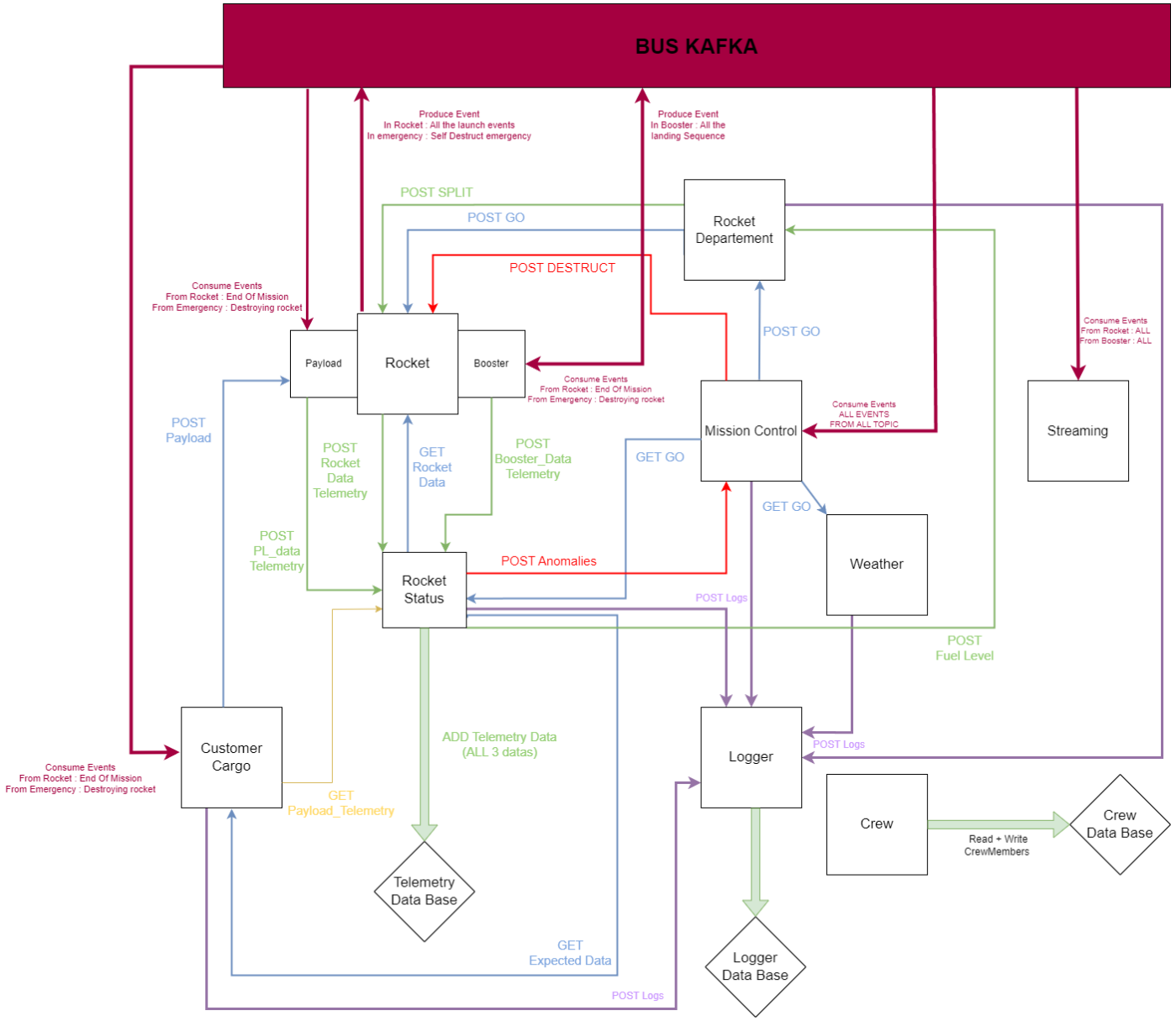
Mars Y

Équipe E

Mohamed MAHJOUB
Thomas FARINEAU
Léo KITABDJIAN
Ludovic BAILET

Polytech Nice Sophia SI5-M2 2023-2024

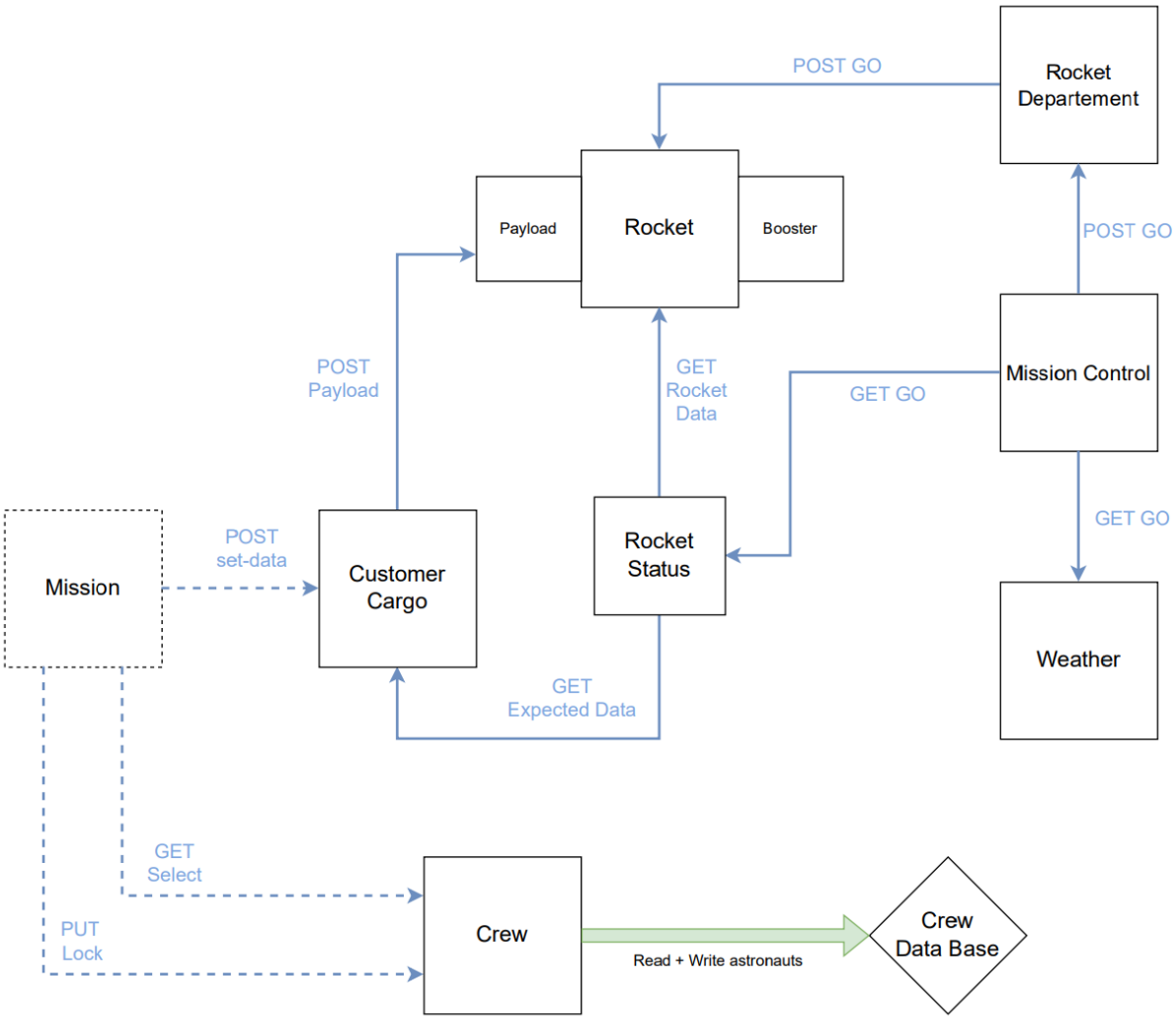
Schéma Architecture Logicielle



Voici le diagramme d'architecture de notre projet. Pour faciliter son explication, nous avons séparé les opérations :

- les Opérations au sol (en bleu)
- les opérations en plein vol concernant l'opération initiale (en vert)
- les opérations secondaires (en rouge et jaune)
- les Opérations de Logs (en violet)

Les opérations au sol



Ces opérations concernent toutes les communications qui se déroulent avant le lancement de la fusée. Afin de mieux les expliquer, nous allons suivre un scénario classique du lancement de la fusée.

- Si nous avions eu l'opportunité de développer les US 20 et 21, nous aurions ajouté un service supplémentaire nommé 'Mission'. Ce service aurait pour rôle de définir le type de mission - 'orbit', 'exploration', ou 'payload' - et de sélectionner les services appropriés en fonction de ce type.
    - Pour une mission 'orbit', qui consiste en l'envoi d'astronautes en orbite avec éventuellement un peu de matériel mais sans charge spécifique, seul le service **Crew** serait sollicité. [User Story 20]
    - Les missions 'exploration', quant à elles, impliquent à la fois des astronautes et un chargement, comme un véhicule (par exemple) destiné à l'exploration d'autres corps célestes, donc les services **CustomerCargo** et **Crew** serait sollicité. [User Story 21]

- Enfin, le 'payload' correspond à notre scénario de base où un envoi est effectué uniquement via le service **CustomerCargo**, selon le scénario décrit ci-dessous.

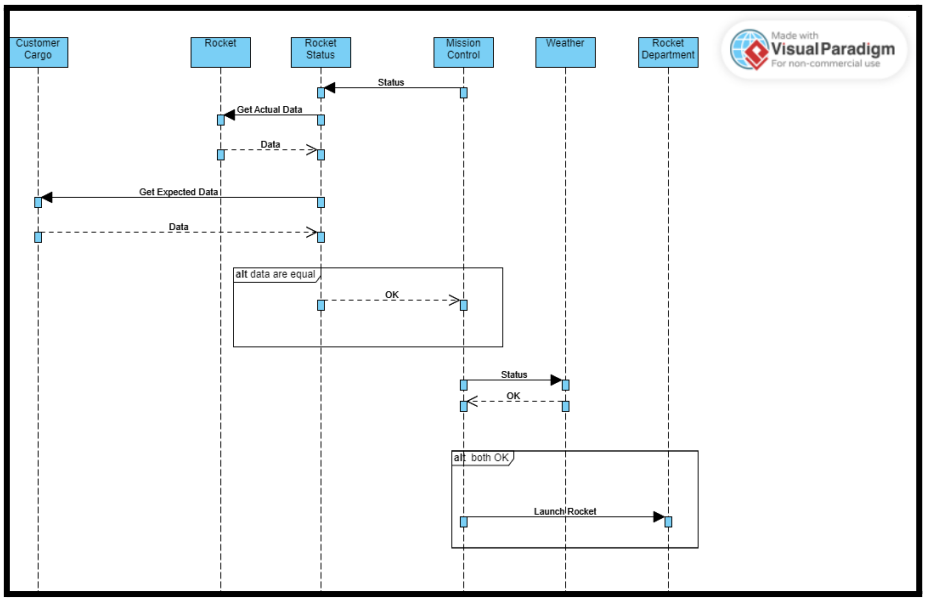
- Avant le début de la mission, le **CustomerCargo** charge le **Payload** à envoyer dans la fusée (avec la requête *POST* **Payload**).

- Le **MissionControl** veut lancer la fusée, il va donc créer un sondage (Poll) et consulte les départements **Weather** et **RocketStatus** pour vérifier si les conditions sont valides pour lancer la fusée. (Requêtes *GET* GO)
  [User Story 3]

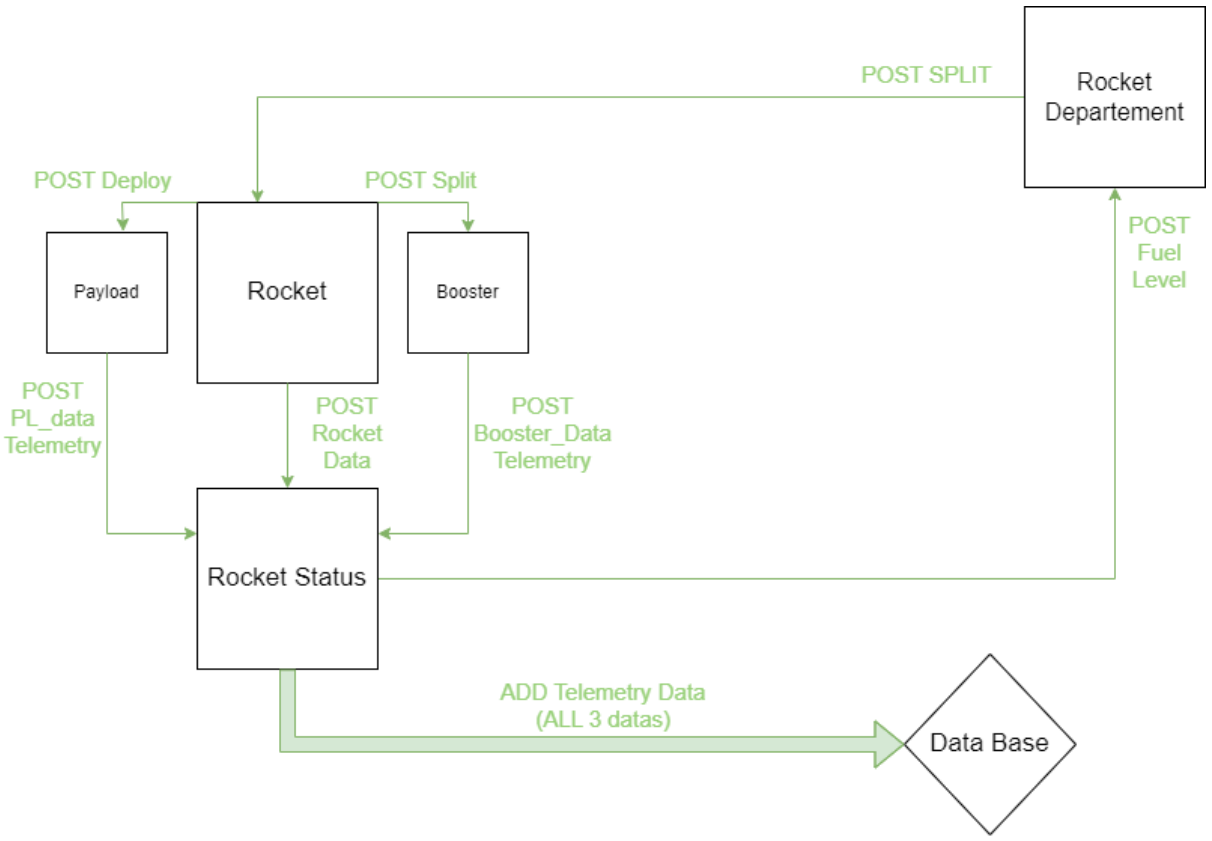
- Le **Weather** se contente de renvoyer GO ou NO GO. (il répond à la requête de **MissionControl)**
  [User Story 1]

- Le **RocketStatuts** afin de répondre à la requête de **MissionControl** va vérifier si la **Rocket** est prête (Il envoie une requête *GET* à la **Rocket)**, de plus il va aussi vérifier le contenu du **Payload** en le comparant avec ce que le **CustomerCargo** souhaite envoyer (à l'aide d'une requête *GET*). Si tout est valide, il renvoie GO au **MissionControl.**
  [User Story 2]

- Une fois le Poll complété (et validé) le **MissionControl** envoie une confirmation au **RocketDepartment** (POST GO) une fois cette confirmation reçue, le **RocketDepartment** va envoyer un *POST* GO à la **Rocket** de démarrer.
  [User Story 4]



Les opérations de vol initiales



Ces opérations concernent toutes les communications qui se poursuivent pendant le lancement de la fusée. Afin de mieux les expliquer, nous allons suivre le scénario du vol avec succès.

- Une fois en plein vol, la **Rocket** envoie continuellement (à l'aide de requêtes *POST*) au **RocketStatus** ses données, celles-ci seront envoyées au service de **Telemetry** (avec requête *POST*.) qui se chargera de les stocker dans une **Data Base**.
  [User Story 5]

- Durant le vol, **RocketStatus** qui reçoit et traite les données de la **Rocket** et du **Booster** va envoyer les informations concernant l'essence contenue dans le **Booster** à **RocketDepartement**, lorsque le niveau d'essence dans le **Booster** atteint un niveau critique, celui-ci envoie la tâche à la **Rocket** de se séparer du **Booster** (avec une requête *POST*).
  [User Story 6]

  *Le **Booster** se met à envoyer ses données au **RocketStatus** pour les sauvegarder à ce stade.*

- Une fois que la **Rocket** a atteint son objectif, il va déployer le **Payload** dans l'espace.
  [User Story 7]

  *Le **Payload** se met à envoyer ses données au **RocketStatus** pour les sauvegarder à ce stade.*

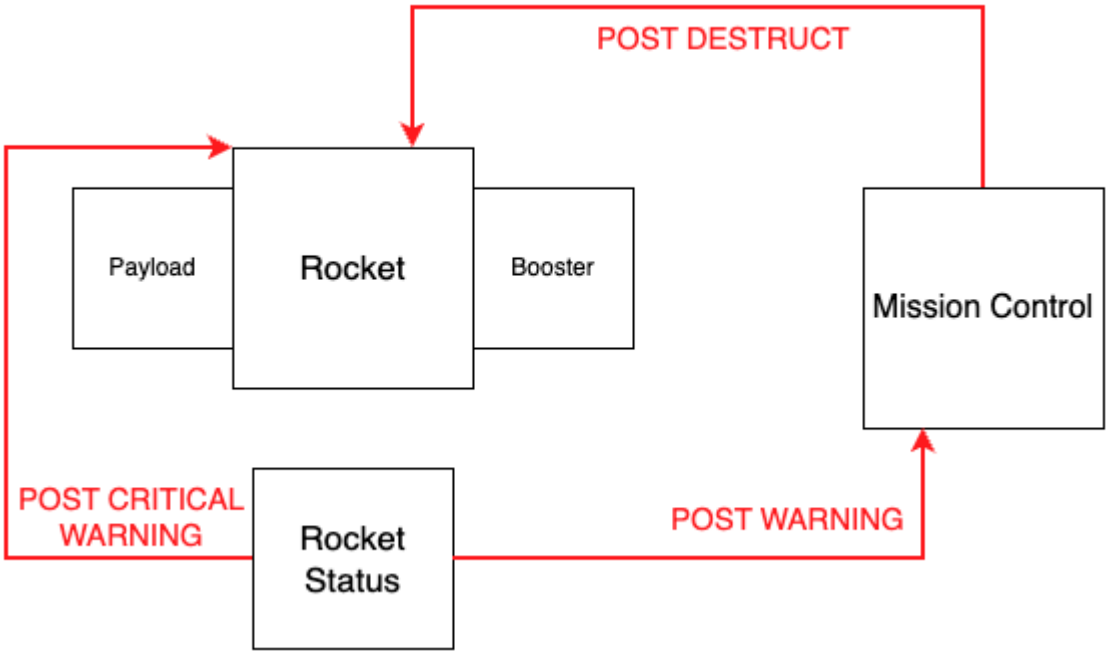
Les différents éléments de la **Rocket** devenus indépendants ont donc désormais leurs propres tâches à effectuer, si l'on part du moment où le **Booster** s'est séparé de la **Rocket** (avec un *POST*)

- Le **Booster** va chercher à atterrir sur le sol de manière contrôlée afin de pouvoir être réutilisé dans le futur.
  [User Story 9]

- De plus, le **Booster** va envoyer sa télémétrie (avec des requêtes *POST*) de façon continue à partir du moment où il s'est détaché au service de **RocketStatus**.
  [User Story 10]

- Le **Payload** lui aussi devient indépendant après son déploiement, il se met à envoyer ses propres données de télémétrie au service de **Rocket Status.**
  [User Story 11]

Les opérations de vol secondaires



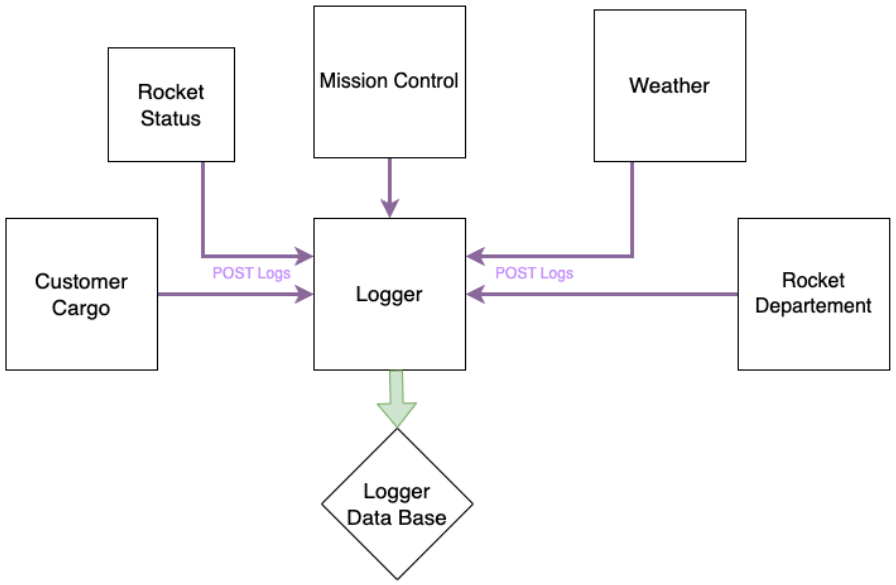
Jusqu'à présent, nous avons supposé que toutes les opérations se déroulaient parfaitement. Cependant, nous devons envisager l'hypothèse de problèmes de trajectoire en plein vol.

Afin de gérer cette éventualité, nous avons mis en place une fonctionnalité répondant à la [User Story 8], qui nous permet de détruire la Rocket en cours d'opération si nécessaire.

Pour satisfaire cette condition, nous utilisons le service **Rocket Status**, qui reçoit en permanence les données de la **Rocket** durant la mission. Grâce à ces données, il est en mesure de déterminer si la trajectoire de la Rocket dévie. Si c'est le cas, il partage cette information avec le **MissionControl** (à l'aide d'une requête *POST*).

Ensuite, la décision de faire exploser la **Rocket** en plein vol, y compris ses composants non déployés, revient à **MissionControl**. Si cette décision est prise, **MissionControl** envoie une demande de destruction à la **Rocket** (à travers une requête *POST*) et celle-ci sera alors détruite, mettant ainsi fin à la mission.

Les opérations de logs

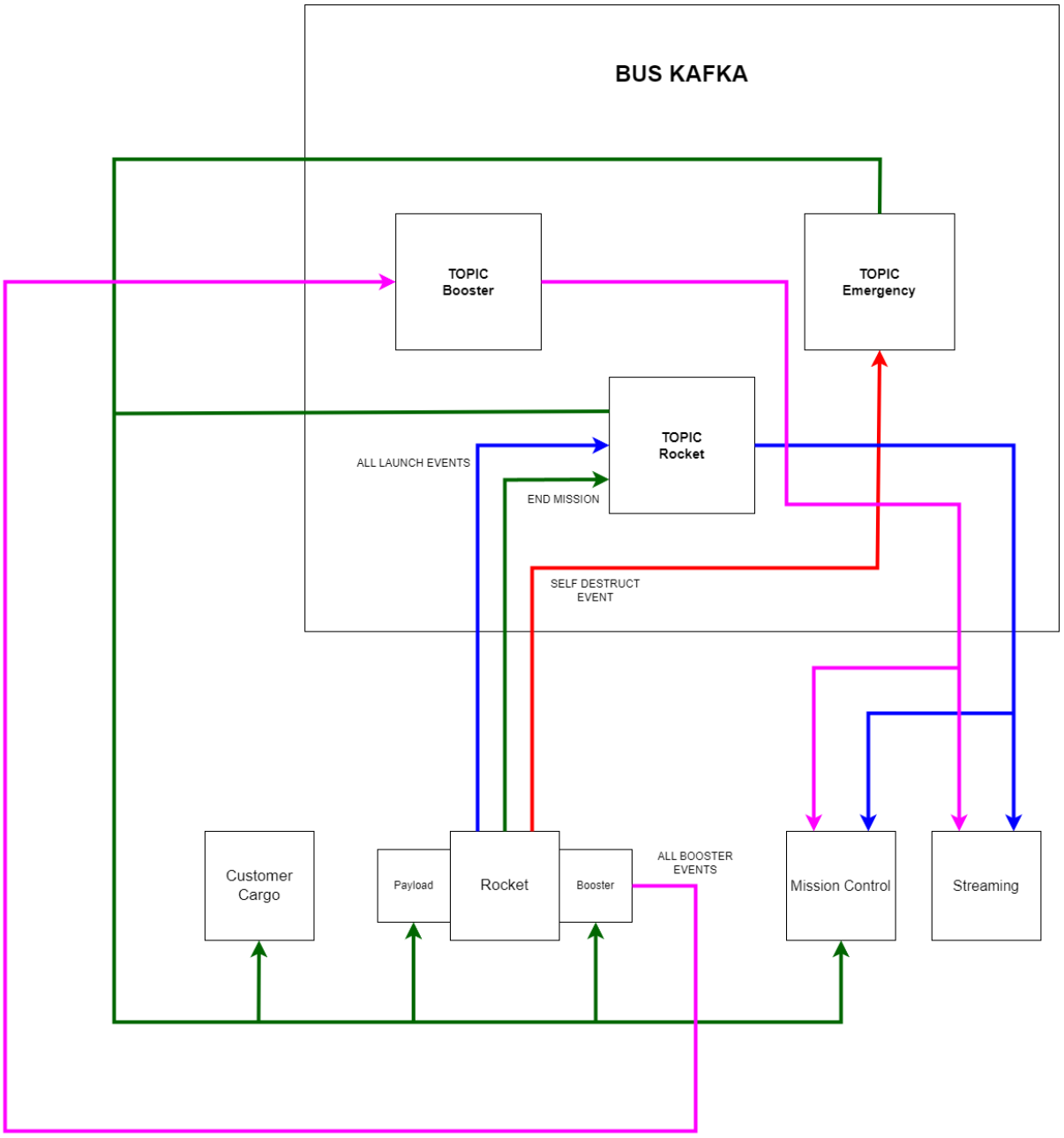


Afin de satisfaire la [User story 14] nous avons mis un service de **Logger** en place. Ce service nous permet de centraliser les logs de tous les services. Une fois envoyé, les logs sont stockés dans une base de données mongo. Après être affichés dans la console du service de **Logger**, ils sont enregistrés.

Dans le cas où on a un problème lors de l'enregistrement, le service renvoie un code 500 et les logs sont alors écrits dans la console du service émetteur.

On a choisi de laisser la télémétrie affichée dans les consoles des services **Payload**, **Booster** et **Rocket** pour faciliter la lisibilité.

L’opération de streaming



Les opérations de streaming sont suivies en direct grâce à l'utilisation d’événements suivis dans des topics KAFKA, les events en Bleu et Magenta représentent les différents événements se déroulant durant la mission, les verts sont pour la fin de la mission et les rouges pour l’auto destruction en cas d’anomalie sévère.

Nouvelles User-Stories

Persona : Emma, la responsable des Ressources Humaines pour les astronautes.
[User Story 19] : En tant qu'Emma, je veux un système automatisé pour sélectionner et attribuer des astronautes aux différentes missions en fonction de leurs compétences et des missions qu'ils acceptent afin de faciliter la gestion des astronautes pour les différentes missions.

Persona: Bill, le directeur des Opérations des Missions Habitées
[User Story 20] : En tant que Bill, je veux pouvoir planifier et exécuter des missions habitées pour envoyer des personnes vers des stations en orbite autour de la Terre, dans le but d'accomplir des objectifs scientifiques et de recherche, maintenir une présence humaine continue dans l'espace et tester de nouvelles technologies dans un environnement spatial.

Persona: Elon Musk, le directeur des Missions d'Exploration de Mars
[User Story 21] : En tant que Elon Musk, je veux pouvoir planifier et orchestrer des missions d'exploration, à destination de Mars ou de la Lune (par exemple), capable d'envoyer à la fois des astronautes et une cargaison, afin d'explorer et étudier ces corps célestes, développer une présence humaine durable dans l'espace, et transporter les équipements nécessaires pour des missions de recherche approfondies et d'établissement de bases.

Récapitulatif des User Stories

US 1	✔
US 2	✔
US 3	✔
US 4	✔
US 5	✔
US 6	✔
US 7	✔
US 8	✔
US 9	✔
US 10	✔

US 11	✔
US 12	✔
US 13	✔
US 14	✔
US 15	✔
US 16	✔
US 17	✔
US 18	✔
US 19	✔
US 20	✖
US 21	✖

Nous n'avons pas eu l'opportunité d'implémenter les nouveaux types de missions incluant des astronautes seuls (US 20) ou des astronautes accompagnés de chargements (payloads) (US 21). Cependant, nous avons réussi à développer un service qui gère de manière automatique tous les astronautes disponibles et qui peut les affecter aux missions appropriées (US 19). Ainsi, tout est en place pour une intégration future des astronautes dans notre système.

Justifications et explications de l'architecture

Nous avons réalisé une architecture basée sur les différents métiers et interfaces des user stories. Les services sont donc ici basés sur les besoins des différentes personnes décrites dans les US. Le **CustomerCargo** par exemple va regrouper tout ce qui concerne la demande du client (envoyer les données attendues, les définir, savoir quand la mission est terminée...). Notre but a été de définir des services s'occupant de sections bien distinctes de la mission et de son déroulement. Le **MissionControl** valide les données avant la mission et est au courant de son déroulement, le **RocketDepartment** sera celui qui donne les ordres à la fusée (démarrer, split le booster, délivrer le payload).

Les trois services **Rocket**, **Payload** et **Booster** sont distincts, car ils correspondent à des équipements qui seront embarqués sur ces équipements, et étant donné qu'ils se séparent dans les nouvelles US, nous sommes partis du principe

qu'ils devaient être indépendants pour assurer leur fonctionnement après les stades de détachement.

Le **RocketStatus** va se charger de gérer les données des éléments de la **Rocket** et de ses équipements afin de gérer la logique métier et d'éviter au maximum la logique à l'intérieur des services embarqués (les trois précédents). Puisqu'il dispose de toutes les données, c'est donc lui qui va les enregistrer et qui va avertir par exemple le rocket department à certains moments (fuel level bas, anomalie, etc...)

Le **Logger** est là pour centraliser tous les logs. Il est chargé de les inscrire dans une base de données (séparée de celle liée à la télémétrie). Il est appelé par pas mal de services mais de par son caractère totalement stateless, il est facilement scalable.

Le service **Streaming** correspond au Webcaster et gère la partie qui avertit le web stream de ce qui se passe. Il est donc là pour consommer les événements kafka liés à la mission dans les topics Rocket et Booster et les enregistrer / restranscrire.

Événements

Les événements générés par notre système suivent les instructions de la [User story 13]. Le service **Rocket** va générer ses propres événements sur le topic Rocket et le service **Booster** les siens sur le topic Booster. Le nom de l'événement sera la "key" et si l'on a besoin d'y rajouter du contenu, on l'inclut dans la "value". Tous les événements des deux topics Rocket et Booster sont consommés par le service **Streaming** ([User story 15]) pour ainsi informer le live stream de l'état de la mission. En revanche, les événements liés à des anomalies par exemple se font dans un topic "emergency" que le service **Streaming** ne consomme donc pas car nous sommes partis du principe que cela ne concerne que les services gérant le bon déroulement de la mission. De plus, nous avons fait en sorte de migrer une partie de l'ancien fonctionnement par requête REST vers une consommation d'événements Kafka.

Par exemple, le setup du **Booster** et du **Payload** précédemment initialisé par une requête REST sont désormais effectués après avoir reçu l'événement *Startup/Launch* de la fusée. De même, l'envoi de télémétrie de la part du **Booster** et du **Payload** s'effectuent dorénavant après avoir reçu respectivement les événements *Stage separation/Deploy* (**Booster**) et *Payload Deploy* (**Payload**) du topic Rocket.

Nous avons décidé de privilégier l'utilisation d'événements, surtout pour les parties les plus importantes de la mission, comme la séparation du booster et du payload. Nous pensons que ces moments sont cruciaux, et nous voulons être sûrs que tout se passe bien et au bon moment. Avec Kafka, qui fonctionne

sur un principe de réactions aux événements (Event-Driven), on est plus sûr que les actions se déclenchent exactement quand elles le doivent.

Kafka nous aide beaucoup dans la gestion des parties **Booster** et **Payload** de nos missions spatiales. C'est mieux que les requêtes REST car Kafka s'occupe tout seul de réessayer si quelque chose ne fonctionne pas au premier coup. Il garde aussi une trace de tous les événements. Cela nous rend plus confiants que tout se passera comme prévu, et cela en temps réel.