



Projets de fin d'études (PFE)

Mémoire de Projet : VISCELL Master 2 Génie Logiciel

AMOSSE Nikolai, ITHURBIDE Martin, SENEL Yusuf, TALBI Simon
LEROY Valentin, LE CORRE Adrien

Encadrants : Philippe Narbel, David Auber, Emmanuel Fleury
Client : Yanis Asloudj

Mars 2024

Table des matières

Table des figures	3
1 Contexte	4
2 Problématique	5
3 Analyse des besoins	6
3.1 Besoins fonctionnels	6
3.1.1 Créer et générer la métaphore visuelle	6
3.1.2 Charger un document et générer sa métaphore visuelle	6
3.1.3 Créer un histogramme de gènes pour chaque population de cellule	7
3.1.4 Colorer les gènes et les liens avec des couleurs fixes distinctes .	7
3.1.5 Ajuster l'opacité des liens du diagramme pour rendre le facteur de confiance visible	7
3.1.6 Afficher les histogrammes au complet lorsqu'un est sélectionné	7
3.1.7 Fournir un hyperlien cliquable pour chaque gène	8
3.1.8 Exporter le diagramme complet	8
3.2 Besoins non fonctionnels	9
3.2.1 Affichage rapide des diagrammes et histogrammes complets . .	9
3.2.2 Accessibilité depuis les navigateurs connus	9
3.2.3 Bonne ergonomie de l'application	10
4 Choix techniques et organisation	11
4.1 Choix techniques	11
4.2 Organisation	12
4.3 Flux opérationnel	12
4.3.1 Feature Branch Workflow	12
4.4 Pipelines CI/CD	16
5 Architecture et description de l'implémentation	18
5.1 Architecture générale	18
5.2 Composants	19
5.2.1 Sankey	19
5.2.2 Barplot	20
5.2.3 FileImport	21
5.2.4 About, Footer, MenuBar, Input	21
5.3 Utils	22

5.3.1	Color	22
5.3.2	Constants	23
5.3.3	SankeyStructure	23
5.3.4	Validation	24
5.4	Diagrammes	25
6	Tests et tactiques	26
6.1	Tests unitaires	26
6.1.1	Tests d'acceptation	28
6.1.2	Tests négatifs	29
6.2	Tests de domaine	30
6.2.1	Tests aux limites	30
7	Analyse des sprints effectués	32
8	Utilisation d'IA génératives	34
9	Améliorations Possibles	35
10	Conclusion	36
A	Maquettes	37

Table des figures

1	Prototype de métaphore visuelle réalisée par le client	5
2	Capture d'écran partielle du Sprint 6 sur Jira	13
3	Noms des branches associées aux tâches du Sprint 6	13
4	Capture d'écran partielle des demandes de fusion	14
5	Capture d'écran regroupant tous les auteurs de commits du projet . .	15
6	Capture d'écran d'un pipeline CI/CD réussi	16
7	Capture d'écran d'un pipeline CI/CD échoué	17
8	Diagramme représentant l'arborescence de l'application	18
9	Diagramme de séquence de l'application	25
10	Squelette d'un code de test React typique	26
11	Capture d'écran du menu listant tous les pipelines du dépôt	27
12	Capture d'écran du résumé de tous les tests d'un commit	27
13	Extrait d'un scénario de test positif du fichier <code>FileImport.test.js</code> .	28
14	Extrait de scénarios de tests négatifs du fichier <code>FileImport.test.js</code> .	29
15	Capture d'écran du diagramme généré depuis le fichier <i>Baron.xlsx</i> . .	30
16	Chronologie des sprints	32
17	Rapport de burndown du premier Sprint	33
18	Rapport de burndown du cinquième Sprint	33
19	Page d'accueil imaginée pour la maquette	37
20	Page d'accueil une fois le fichier soumis	38
21	Page À propos	38
22	Page d'accueil avec le diagramme d'exemple	39

1 Contexte

Notre projet porte sur le domaine de la bio-informatique. Ce domaine est un juste milieu entre l'informatique, la biologie et les statistiques, visant à développer des méthodes novatrices en ce qui concerne l'analyse des données biologiques.

Dans notre cas, nous étudions plus particulièrement les **cellules**, ainsi que les gènes qui les composent. Les cellules chez l'être humain se différencient en divers *types cellulaires* pour exécuter des fonctions essentielles.

Par exemple, les neurones transmettent des signaux électriques et les globules blancs combattent les agents pathogènes. Chaque type cellulaire peut être caractérisé par un ensemble de **gènes** qui lui sont spécifiques.

La technologie de *single-cell* représente une avancée révolutionnaire en permettant la mesure de l'activité des gènes au sein de chaque cellule d'un échantillon, comme une tumeur par exemple.

En utilisant des algorithmes de regroupement, ces cellules peuvent être classées en **populations** homogènes et caractérisées en fonction des gènes qu'elles expriment de manière spécifique.

Ainsi, pour interpréter les résultats d'une analyse *single-cell*, il est nécessaire de concevoir des métaphores visuelles qui intègrent toutes les informations pertinentes concernant les populations cellulaires telles que leur taille, les gènes qu'elles expriment et leur fiabilité.

2 Problématique

De ce constat, notre **client** Yanis Asloudj, doctorant en bio-informatique, a imaginé une métaphore visuelle innovante permettant de représenter à la fois une famille de cellules via un diagramme de Sankey [1], mais également les expressions des gènes au sein même de chaque cellule via un histogramme.

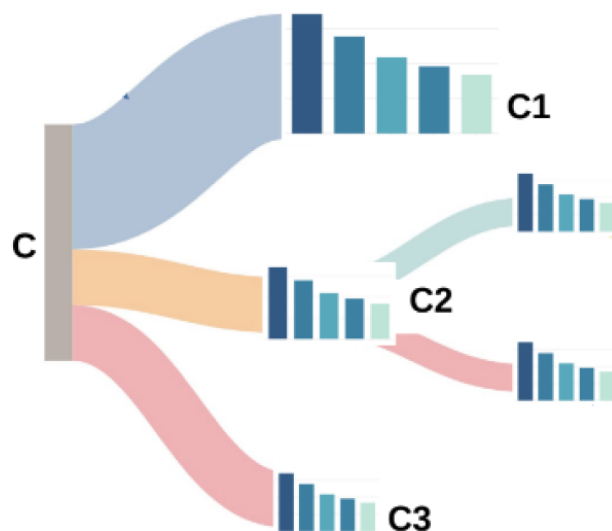


FIGURE 1 – Prototype de métaphore visuelle réalisée par le client

Avec ça, nous avons reçu des directives claires de la part du client. Il souhaite une application web permettant de générer la représentation qu’il a imaginée à partir d’un ou plusieurs fichiers de données conformes à une norme qu’il a établie. Cette norme n’est pas vouée à changer de si tôt et est issue des résultats de son travail de *clustering* qu’il entreprend au LaBRI. Aucune technologie ou framework n’a été imposé.

3 Analyse des besoins

De nos discussions fréquentes, nous avons pu dresser un cahier de besoins fonctionnels et non fonctionnels afin de mener ce projet à bien.

3.1 Besoins fonctionnels

Les besoins fonctionnels décrivent les fonctionnalités ainsi que les services qui seront intégrés dans notre application, ceux qui sont **nécessaires** au projet. Les besoins ci-dessous sont rangés par ordre de priorité.

3.1.1 Créer et générer la métaphore visuelle

C'est un besoin **essentiel**. La métaphore visuelle imaginée par le client se place à la première place en terme de priorité. En effet, le client souhaite que son idée prenne vie, au travers de son application. Pour ce besoin, il est impératif de pouvoir afficher un diagramme de Sankey, avec comme racine un rectangle représentant la population initiale nommée C , et le reste des enfants par des histogrammes.

3.1.2 Charger un document et générer sa métaphore visuelle

Ce besoin est également **essentiel**. Il est nécessaire pour le client de pouvoir obtenir sa métaphore visuelle provenant de **ses données**. Après diverses opérations de *clustering* réalisées sur un Jupyter Notebook, il génère facilement un fichier au format *XLSX* contenant deux feuilles :

- La feuille *meta* contenant des informations sur les liens de parenté des populations, ainsi que la valeur de confiance qui leur est attribuée selon la métrique F_1 Score ou *moyenne harmonique* [2]. Cette feuille est essentielle pour la génération du diagramme.
- La feuille *markers* quant à elle met en rapport tous les gènes avec toutes les populations, et plus particulièrement si un gène est présent, et plus ou moins exprimé dans une population précise. Cette feuille est primordiale pour la génération des histogrammes.

3.1.3 Créer un histogramme de gènes pour chaque population de cellule

C'est un besoin **essentiel**. Ces petits histogrammes représentant des populations de cellules sont en fait les enfants de la population initiale dans le diagramme de Sankey. En recherche scientifique, trois gènes sont généralement nécessaires pour identifier une pathologie ou une maladie.

Nous avons d'ailleurs poussé l'exploration de ce besoin encore plus loin et sommes désormais capables d'afficher de manière dynamique entre 3 et 7 enfants dans ces histogrammes réduits. La limite de 7 a été fixée par le client, au-delà les autres gènes deviennent moins pertinents à observer en raison de leur faible expression.

3.1.4 Colorer les gènes et les liens avec des couleurs fixes distinctes

Nous développerons un peu plus sur l'algorithme de coloration des liens et des gènes, mais ce besoin est **essentiel** à la réussite du projet. L'intérêt de cette métaphore visuelle est de donner une couleur fixe aux gènes dits *spécifiques* et également de représenter les familles et sous familles de population avec ces couleurs.

3.1.5 Ajuster l'opacité des liens du diagramme pour rendre le facteur de confiance visible

Nous avons d'ores et déjà parlé du facteur de confiance que l'on accorde à une population. Le facteur de confiance, attribué à une population après un processus de *clustering*, établit une corrélation entre diverses valeurs associées à cette population à travers de nombreux processus de *clustering*. Il indique le degré de confiance quant à la fiabilité des résultats obtenus, permettant d'évaluer la certitude dans la mesure des données et des conclusions tirées. Il est donc **essentiel** de pouvoir représenter cette valeur en variant l'opacité des liens d'un nœud parent à un nœud enfant.

3.1.6 Afficher les histogrammes au complet lorsqu'un est sélectionné

C'est un besoin **essentiel**. Chaque population possède un nombre indéfini de gènes exprimés à l'intérieur, nous devons donc être en mesure de visualiser ces informations rapidement, à l'intérieur d'un histogramme complet et gradué.

3.1.7 Fournir un hyperlien cliquable pour chaque gène

Ce besoin est considéré comme **essentiel**. Comme résumé précédemment, nous disposons de gènes distincts et identifiables par un nom ainsi qu’une couleur. Ces gènes ont été répertoriés au fil des années et des recherches dans plusieurs bases de données.

Notre client souhaite accéder aux informations de n’importe quel gène simplement en un clic, en accédant aux informations publiées par la *United States National Library of Medicine* (NLM) grâce à un hyperlien, paramétrable grâce au nom du gène [3]. Ces gènes concernent uniquement l’espèce *Homo sapiens*.

3.1.8 Exporter le diagramme complet

Un autre besoin se classant cette fois-ci dans la catégorie des besoins **conditionnels** – c’est-à-dire des besoins qui étendent et améliorent le logiciel – est celui d’exporter le diagramme. En effet, dans le domaine scientifique, il est nécessaire de coopérer pour comparer et améliorer nos résultats.

C’est dans cette optique que l’export de la métaphore visuelle au complet devient intéressante. Notre client veut par exemple être en mesure de générer une image vectorielle de la métaphore – au format *SVG* – et sans perte de qualité. Il veut notamment utiliser la commande `\usepackage{svg}` afin de facilement insérer ce format d’image dans des articles de recherche écrits en L^AT_EX.

En suivant les conseils des encadrants de projets de fin d’études, nous avons également développé la fonctionnalité d’export du diagramme en *PDF*, garantissant un partage facile pour ceux n’étant pas familiers avec les formats d’images scalables.

3.2 Besoins non fonctionnels

Les besoins non fonctionnels quant à eux précisent les qualités, les contraintes, ainsi que les attributs que l'on attend du logiciel et de son fonctionnement.

3.2.1 Affichage rapide des diagrammes et histogrammes complets

Il est primordial que l'application soit capable d'afficher rapidement la métaphore visuelle voulue par le client. Le chargement des fichiers et la génération de la métaphore doit donc prendre un temps raisonnable même si aucun travail d'édition ne rentre en jeu après. Ce besoin est justifié car un affichage rapide, même dans le cas d'une simple visualisation, est crucial pour assurer l'efficacité de parcours d'un grand jeu de données. Une partie non négligeable du temps de travail sera cependant réservée à l'amélioration des complexités des algorithmes de vérification, de génération mais également de coloration.

3.2.2 Accessibilité depuis les navigateurs connus

Les navigateurs web couramment utilisés, en particulier ceux utilisant le moteur de rendu *Chromium*, doivent pouvoir accéder et utiliser notre application sans problème, en pouvant tirer pleinement parti de toutes ses fonctionnalités. Cette exigence garantit que le plus grand nombre d'utilisateurs puisse se servir de notre outil sans se soucier de quelconques problèmes de compatibilité. Notre logiciel est probablement voué à évoluer, et contribuer à son accessibilité et sa pertinence facilitera son adoption par un large éventails de chercheurs, enseignants, et même étudiants.

Nous précisons le besoin en parlant des navigateurs tournant sous *Chromium* car d'autres comme Firefox ont toujours quelques problèmes quant au *rendering* des composants vectoriels.

3.2.3 Bonne ergonomie de l'application

Ce besoin non fonctionnel découle du manque de conventions concernant l'approche de *clustering* en *single-cell*. L'application doit être non seulement facilement compréhensible par un large public, mais aussi efficace et agréable à utiliser, surtout si elle est destinée à devenir un outil de travail. Les utilisateurs doivent pouvoir atteindre leurs objectifs sans difficulté, en un minimum de clics et en respectant les normes actuelles en matière d'expérience utilisateur (UX).

Dans notre cas, de nombreux curseurs différents doivent être affichés, ainsi que des effets visuels lorsqu'on survole des éléments importants, entre autres. Il est également essentiel que toutes les actions soient claires, avec des bulles d'information disponibles pour guider les utilisateurs.

4 Choix techniques et organisation

Dans cette section, nous allons aborder en détail les choix que nous avons faits concernant la conception, la création et le développement de l’application, en mettant en lumière les considérations et les décisions qui ont guidé notre travail.

4.1 Choix techniques

Notre client exige que son logiciel prenne la forme d’une **application web**. Ce choix est en effet judicieux pour deux principales raisons. Tout d’abord, l’accessibilité et la simplicité d’utilisation offertes par une application web correspondent parfaitement aux besoins de notre client, et par la suite de nos potentiels utilisateurs. Ensuite, cette décision a été renforcée par la disponibilité de nombreuses librairies efficaces, spécifiquement conçues pour résoudre les problèmes auxquels nous étions confrontés.

Pour cela, nous avons étudié et expérimenté différentes bibliothèques JavaScript dédiées à la manipulation et la visualisation de données à travers des diagrammes de Sankey, telles que *react-google-charts* [4], *plotly* [5] ou encore *D3.js* [6].

Après avoir effectué divers tests afin de mieux appréhender les avantages et les limitations de chaque bibliothèque, notre préférence s’est orientée vers **D3.js**. Cette décision découle de sa modularité plus avancée par rapport aux autres solutions examinées, nous offrant la praticité et la flexibilité nécessaire pour personnaliser les diagrammes créés sans être contraints par des modèles préexistants et peu adaptables.

En raison de la nature fondamentale du besoin de ce projet et après avoir consulté notre client, nous sommes parvenus à la conclusion qu’une application cliente sans serveur serait suffisante. L’application ayant pour seule vocation de lire un fichier local et afficher une métaphore visuelle, la présence d’un serveur n’est pas nécessaire.

À ce sujet, notre client a vivement souligné l’importance de la documentation, de la clarté du code, ainsi que sa maintenabilité et son extensibilité. Dans le futur, il veut être à même de pouvoir intégrer d’autres modules dans le projet, tels qu’un côté serveur par exemple. Les stratégies adoptées pour répondre à cette demande seront détaillées tout au long du rapport.

Par conséquent, notre choix s’est naturellement porté sur le framework JavaScript **React** [7], en parfaite adéquation avec notre vision d’une application client. React, qui, comme d’autres frameworks tels que Vue ou Angular, propose une approche où chaque élément est conçu comme un composant, nous offre une certaine modularité nécessaire dans ce projet. Cette approche permet à notre équipe de concevoir et de structurer l’application de manière hautement adaptable et évolutive. De plus, le fait que nous ayons tous une expérience préalable avec React a conforté notre choix de ce dernier par rapport aux autres frameworks disponibles.

4.2 Organisation

En ce qui concerne l’organisation de notre travail, nous avons rapidement opté pour la mise en place de réunions hebdomadaires avec notre client. Cette approche nous a permis de planifier nos sprints sur une base hebdomadaire, avec des objectifs à court terme en accord avec les attentes de notre client et l’ensemble de l’équipe de développement.

Après deux ou trois sprints, nous avons réalisé qu’il était nécessaire d’augmenter la fréquence de nos réunions. Cela était motivé à la fois par le développement d’algorithmes complexes et notre volonté de ne pas faire d’erreurs, mais aussi par l’intérêt accru des supérieurs de notre client pour l’avancement et la qualité de la métaphore visuelle. Par conséquent, nous avons renforcé nos échanges pour faire le point de manière plus régulière.

4.3 Flux opérationnel

Dans cette section, nous allons rapidement décrire la manière dont notre équipe a travaillé ensemble, les outils et technologies utilisées pour y parvenir, et enfin rapidement parler des conventions que nous avons instauré.

4.3.1 Feature Branch Workflow

Afin de maximiser nos chances de maintenir un développement continu et linéaire, nous avons mis en place une méthode de travail organisée pour permettre de réparer efficacement et équitablement la charge de travail. Pour cela, nous avons décidé d’appliquer le *feature branch workflow* [8], c’est-à-dire travailler sur de nouvelles fonctionnalités de manière isolée, en créant des branches dédiées à chaque fonctionnalité ou à chaque changement important.

De cette manière, nous maintenons notre dépôt **GitLab** clair et lisible en le synchronisant avec nos tâches sur Jira. Les **identifiants** des tâches sur Jira sont utilisés pour nommer les branches, garantissant une traçabilité claire entre les fonctionnalités. Pour mieux illustrer cela, nous vous invitons à jeter un coup d'œil par vous-même :

<input checked="" type="checkbox"/> VISCELL-73	Entourer les populations avec la couleur du lien	DIAGRAMME ...	DONE ▾	3	ST
<input checked="" type="checkbox"/> VISCELL-77	Ajouter dans le tooltip d'un gène à quelle population il ...	HISTOGRAMME	DONE ▾	3	ST
<input checked="" type="checkbox"/> VISCELL-78	Compléter le README avec toutes les commandes	LIVRABLE	DONE ▾	1	VL
<input checked="" type="checkbox"/> VISCELL-74	Changer dynamiquement la taille des tooltips lorsqu'on...	DIAGRAMME ...	DONE ▾	3	VL
<input checked="" type="checkbox"/> VISCELL-76	Ajouter le padding sur les barres des populations	HISTOGRAMME	DONE ▾	1	NA
<input checked="" type="checkbox"/> VISCELL-82	Vérifier le F1 Score et améliorer les Toasts associés	GESTION DES ...	DONE ▾	5	VL
<input checked="" type="checkbox"/> VISCELL-80	Créer un script pour lancer l'application sous Linux	LIVRABLE	DONE ▾	1	MI
<input checked="" type="checkbox"/> VISCELL-83	Rajouter un avertissement pour les fichiers considérés c...	GESTION DES ...	DONE ▾	1	VL
<input checked="" type="checkbox"/> VISCELL-79	Compléter les fonctionnalités dans la page 'À propos'	PAGE 'À PROP...	DONE ▾	1	AC
<input checked="" type="checkbox"/> VISCELL-75	Modifier les ayant droits et rajouter les noms des dével...	PAGE 'À PROP...	DONE ▾	1	AC
<input checked="" type="checkbox"/> VISCELL-72	Parfaire la scalabilité du diagramme de Sankey	DIAGRAMME ...	DONE ▾	8	NA
<input checked="" type="checkbox"/> VISCELL-70	Supprimer la page d'exemple et implémenter le bouton...	DIAGRAMME ...	DONE ▾	3	MI

FIGURE 2 – Capture d'écran partielle du Sprint 6 sur Jira

Branch Name	Commit Hash	Description	Status	Buttons
dev	6a69ec46	(test) added tests for sankey.js and fileimport.js · 11 hours ago	1/44 ✓	New, Download, More
viscell-70	eca40905	(fix) wrote css again after old merge deleted it · 16 hours ago	1/42 ✓	New, Download, More
viscell-79	09731084	Merge branch 'dev' into 'viscell-79' · 17 hours ago	1/36 ✓	New, Download, More
viscell-72	f63ff828	(fix) deletion of width's links scalability · 19 hours ago	1/26 ✓	New, Download, More
viscell-83	eac2005b	(feat) added critical alert and changed dependencies · 1 day ago	1/19 ✓	New, Download, More
viscell-82	65cd3379	Merge branch 'dev' into viscell-82 · 1 day ago	1/15 ✓	New, Download, More
viscell-76	952eb50f	Merge branch 'dev' into viscell-76 · 1 day ago	1/9 ✓	New, Download, More

FIGURE 3 – Noms des branches associées aux tâches du Sprint 6

Durant le développement, nous avons préservé un esprit de collaboration et d'entente en s'appelant régulièrement, bien souvent pour faire du *pair programming* ou pour faire des *code reviews* lorsque le besoin se faisait sentir.

En plus des branches dédiées aux fonctionnalités, nous avons la branche *main*, mise à jour chaque lundi pour servir de livrable marquant la fin de chaque sprint, ainsi que la branche de développement sobrement nommée *dev*.

De ce fait, nous avons un certain nombre de *merge requests*, que nous avons toujours fait apparaître sur Git. En effet, le développement étant continu sur la branche *dev*, nous fusionnons dessus dès que les branches contenant des fonctionnalités achevées sont prêtes. Parfois, il est également nécessaire de mettre à jour une branche de fonctionnalité en faisant un *rebase* si elle prend trop de retard par rapport à *dev*.

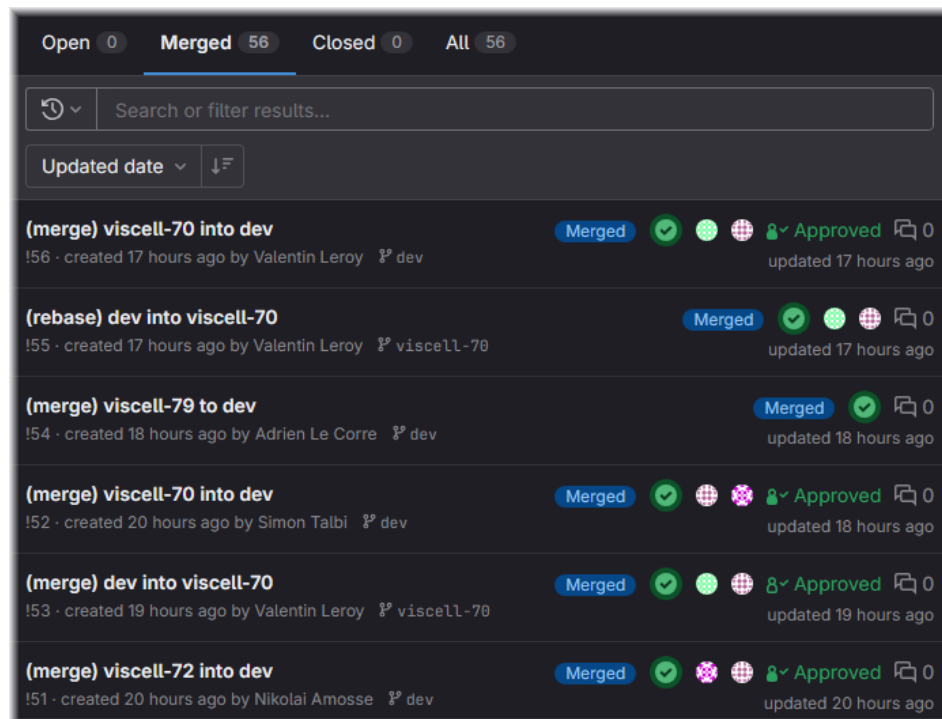
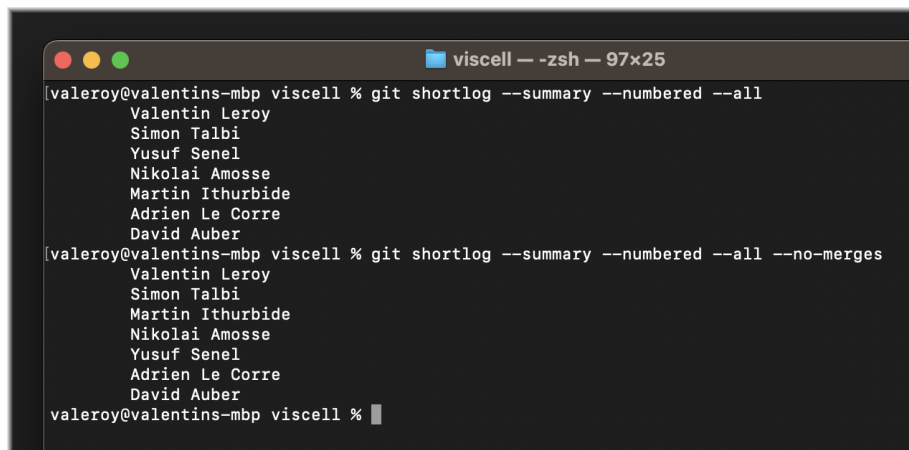


FIGURE 4 – Capture d'écran partielle des demandes de fusion

De plus, chaque message de *commit* est fait en respectant des règles que nous nous sommes élaborées au début du projet. Tout d'abord, nous avons fait le choix

d'écrire tous nos message de commits en anglais. Avant chaque message de commit, nous précisons de quel type de commit il s'agit entre parenthèses avant le message, (*feat*) si c'est une nouvelle fonctionnalité, (*fix*) si nous avons corrigé un bug, (*test*) si nous avons ajouté des tests unitaires, (*tech*) s'il s'agit d'une modification technique (pipeline, librairies, etc), (*merge*), (*rebase*).

Nous remarquons également, à l'aide des commandes suivantes, qu'il n'y a pas de doublons parmi les auteurs des commits ni d'erreurs de *credentials*. Ces vérifications peuvent également être effectuées dans l'onglet *Analyze*, sous la section *Contributor analytics*, sur notre dépôt Git. De plus, nous avons veillé à maintenir une convention cohérente dans nos messages de commits.



```
viscell — -zsh — 97x25
[valeroy@valentins-mbp viscell % git shortlog --summary --numbered --all
    1 Valentin Leroy
    2 Simon Talbi
    3 Yusuf Senel
    4 Nikolai Amosse
    5 Martin Ithurbide
    6 Adrien Le Corre
    7 David Auber
[valeroy@valentins-mbp viscell % git shortlog --summary --numbered --all --no-merges
    1 Valentin Leroy
    2 Simon Talbi
    3 Martin Ithurbide
    4 Nikolai Amosse
    5 Yusuf Senel
    6 Adrien Le Corre
    7 David Auber
valeroy@valentins-mbp viscell %
```

FIGURE 5 – Capture d'écran regroupant tous les auteurs de commits du projet

4.4 Pipelines CI/CD

Les pipelines CI/CD (Continuous Integration/Continuous Development) sont des pratiques essentielles dans le développement logiciel moderne, visant à automatiser les processus de construction, de test et de déploiement d'applications. Dans notre projet, nous avons mis en œuvre un pipeline CI/CD en utilisant GitLab CI/CD.

Notre fichier de configuration `.gitlab-ci.yml` définit deux étapes principales : la construction (*build*) et les tests unitaires (*test*). Pour la construction, nous utilisons une image Docker avec la dernière version de Node.js. Avant d'exécuter le script de construction, nous installons les dépendances nécessaires en utilisant la commande `npm install` au sein du répertoire *client*. Le script de construction lui-même consiste à exécuter la commande `npm run build` pour générer les artefacts de l'application.

Quant aux tests, nous suivons un processus similaire en utilisant également une image et en installant les dépendances nécessaires avant d'exécuter les tests. Nous utilisons la commande `npm run ci`, qui exécute les tests en utilisant *Jest* [9] grâce au processeur de résultats *jest-junit*. Cela permet de générer des rapports de tests au format JUnit, facilitant ainsi l'intégration avec d'autres outils. En plus de la génération de rapports, cette commande assure par ailleurs que les tests sont exécutés en mode CI, avec la couverture de code activée, ce qui garantit une évaluation approfondie de la qualité du code. Les rapports de couverture de codes sont ensuite archivés en tant qu'artefacts, fournissant une visibilité sur la qualité du code à chaque itération du pipeline CI/CD.

Enfin, nous avons spécifié des *tags* pour exécuter les jobs, utilisant les *shared runners* mis à la disposition par le GitLab du CREMI, garantissant ainsi leur exécution dans un environnement approprié et partagé.

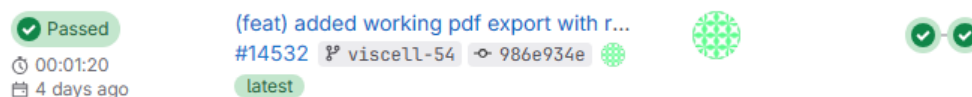


FIGURE 6 – Capture d'écran d'un pipeline CI/CD réussi

Nous pouvons remarquer sur la FIGURE 6, que nous avons bien deux *jobs*, le premier pour la compilation et génération d'une version de production optimisée, et le deuxième pour les tests unitaires. Si un des deux *jobs* échoue, ils auront alors

l'état *Failed* comme en atteste la FIGURE 7 et nous pourrions aller vérifier les logs du pipeline pour trouver l'erreur.

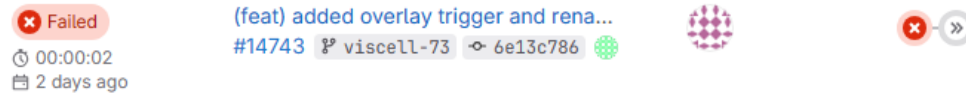


FIGURE 7 – Capture d'écran d'un pipeline CI/CD échoué

Attention ! Nous utilisons l'outil CI/CD de manière stricte. Le pipeline de *build* ne passera pas si nous avons ne serait-ce qu'un seul *warning* dans notre code, par exemple des variables initialisées, mais jamais utilisées. De la même logique, le pipeline de tests ne passera pas si les tests unitaires ne sont pas bons.

5 Architecture et description de l'implémentation

5.1 Architecture générale

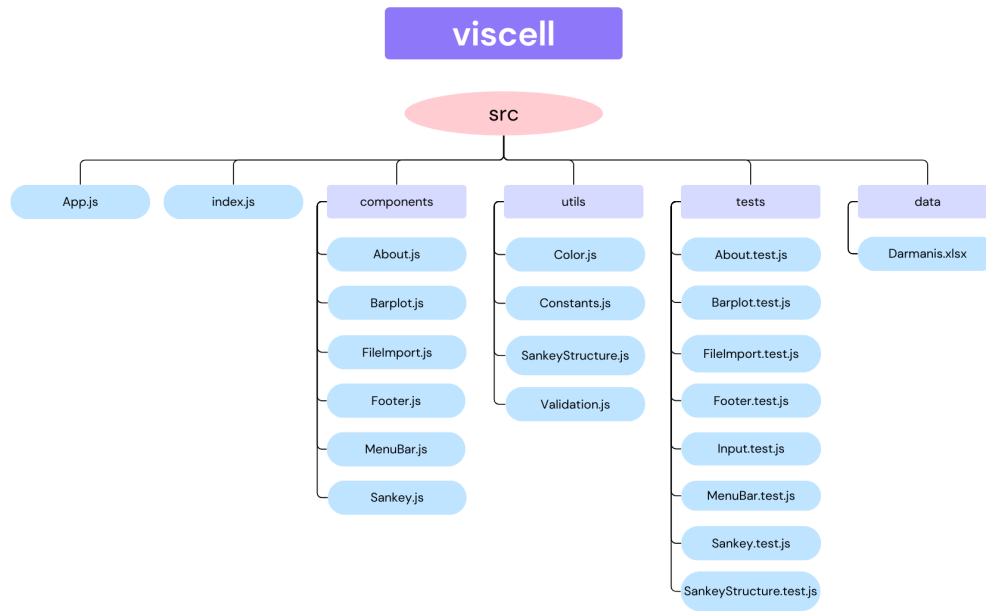


FIGURE 8 – Diagramme représentant l'arborescence de l'application

L'architecture de notre projet est habituelle d'une application React. Vous trouverez dans le répertoire *src* de l'application tous le code source de cette application. Les composants React, fragments de codes modulables et réutilisables, sont placés dans le répertoire *components*.

Le répertoire *utils*, contient quant à lui les fichiers possédant uniquement des fonctions ou des classes qui ne représentent donc pas de composants. Il peut s'agir d'algorithmes précis ou bien de structures de données, que nous utilisons et manipulons dans un composant quelconque. Nous avons séparé ces fichiers pour les différencier des composants, offrant ainsi une meilleure lisibilité et permettant un débogage plus efficace.

Le répertoire *data* contient les fichiers d'exemple au format *XLSX* appelés *Darmanis.xlsx* [10] et *Baron.xlsx* [11]. Enfin, le répertoire *tests* contient les différents fichiers de tests unitaires de l'application.

5.2 Composants

5.2.1 Sankey

Le composant Sankey est le composant utilisé pour afficher le diagramme de Sankey. Nous utilisons la librairie **D3.js** pour créer ce dernier. Pour cela, nous devons renseigner une liste de *Links* et de *Nodes*, chacun étant des objets définis par la librairie. De plus, nous devons initialiser un *layout*, définissant l'apparence du diagramme. Avec ces données, le Sankey peut ensuite être personnalisé avec différents éléments pour chaque type de données, que cela soit des *nodes*, des *links* ou bien même des rectangles, etc. Cela permet de gérer, entre autres, leur couleur, leur épaisseur, leur position (relative ou absolue), et bien plus encore.

D'après le cahier des besoins du projet, chaque nœud du diagramme de Sankey doit être représenté par un histogramme. Cela est possible avec **D3.js**. En effet, bien que les nœuds du Sankey soient, par défaut de simples rectangles de couleur, il est possible d'utiliser du code HTML personnalisé pour les afficher. Ainsi, grâce à une balise HTML *foreignObject*, il nous a été possible d'afficher les composants **React** des histogrammes à la place des nœuds du diagramme. De plus, les contours de ces nœuds sont colorés avec une couleur spécifique à cette population (générée dans *Color.js*).

Pour représenter le consensus, qui indique le degré de confiance relatif à la fiabilité des résultats obtenus, nous utilisons l'opacité des liens du diagramme de Sankey. Plus le consensus liant deux populations de cellules est élevé, plus l'opacité est importante. Cependant, dans des soucis de lisibilité des couleurs choisies, nous limitons l'opacité minimale à 0.2 et l'opacité maximale à 0.9 et effectuons une scalabilité pour rendre les valeurs plus parlantes. En effet, certains consensus peuvent avoir une valeur nulle mais leur lien doit tout de même apparaître.

Pendant le développement, avant que nous implémentions la palette de couleur, les couleurs des liens n'étaient pas assez contrastés. Ainsi, lorsque l'opacité variait, cela rendait les liens de teintes similaires peu différenciables entre eux. Pour remédier à cela, nous avons pensé à une autre façon de représenter cette valeur de consensus sur les liens. Cette deuxième version consistait à faire varier l'épaisseur du contour

d'un lien en fonction de sa valeur de consensus. Après avoir développé une version test que nous avons présentée au client, nous avons abandonné cette idée qui nous semblait peu esthétique. De plus, en comparaison avec la technique de l'opacité, la technique de contours était moins lisible et compréhensible.

Après ce test peu fructueux, nous étions donc convaincus qu'il nous fallait continuer le développement avec la technique initiale de l'opacité des liens, mais en améliorant le choix des couleurs. Ainsi, nous avons utilisé une palette de couleur contrastée améliorant la clarté du diagramme.

Étant donné la nature arborescente du diagramme de Sankey, nous avons créé une classe intermédiaire nommée `SankeyStructure` (présente dans *utils*) de façon à stocker, à la manière d'un arbre, les données lues sur le fichier d'entrée. Grâce à cette structure, nous pouvons remplir les listes `Nodes` et `Links` du diagramme de Sankey de manière très simple. De plus, cela nous permet d'obtenir facilement, une fois dans le composant `Sankey`, les relations parents enfants entre les différents nœuds du diagramme, sans avoir à parcourir les liens de ce dernier.

Nous utilisons aussi la fonction `Color` (présente dans *utils*) afin de générer une palette de couleur et de colorer les populations de cellules et leurs gènes selon la spécificité d'un gène à une population. Cette palette de taille définie est générée grâce à la librairie `D3`.

5.2.2 Barplot

Le composant `Barplot` est associé aux nœuds du `Sankey`, représentant les histogrammes des populations de cellules. Chaque nœud du `Sankey` est rendu dans une balise HTML *foreignObject*, et c'est ce composant `Barplot` qui est lié à chaque *foreignObject*. Le `Barplot` est une composante `React` qui exploite la puissance de la bibliothèque **D3.js**, de manière semblable au composant `Sankey`, pour produire des histogrammes interactifs.

Lorsqu'on utilise le composant `Barplot`, plusieurs paramètres doivent être fournis. Ces paramètres incluent la largeur et la hauteur du graphique, le nom de la cellule étudiée, les données sur les gènes à visualiser, ainsi que des tables de couleurs permettant d'associer chaque gène à une couleur spécifique, et pour représenter les cellules avec différentes couleurs. Ces paramètres sont transmis au `Barplot` lors de son appel depuis le composant `Sankey`.

L'histogramme est généré dynamiquement en fonction des données fournies avec la possibilité de limiter le nombre de gènes affichés. Les gènes sont classés et affichés en fonction de leur importance dans la cellule.

Barplot est également interactif. En survolant une barre, une infobulle s'affiche, indiquant le nom du gène ainsi que des informations supplémentaires sur sa spécificité par rapport à une certaine population cellulaire. De plus, en cliquant sur une barre, l'utilisateur est redirigé vers une page externe où il peut obtenir plus d'informations sur le gène correspondant.

En outre, ce composant inclut une nouvelle fenêtre qui permet d'afficher un histogramme en entier, offrant une vue plus détaillée des données.

5.2.3 FileImport

Comme son nom l'indique, le composant FileImport permet à l'utilisateur d'importer un jeu de données pour pouvoir réaliser sa métaphore visuelle. De plus, une vérification des données est effectuée au moment du chargement du fichier, et rapporte si le fichier est validé ou refusé. Un exemple de fichier non valide est un fichier où il manque une des deux (ou les deux) pages essentielles qui sont *meta* et *markers*. Une infobulle, comportant toutes les raisons pour lesquelles le fichier est refusé sera affiché en cas d'*upload* non concluant.

5.2.4 About, Footer, MenuBar, Input

Les composants About, Footer, MenuBar et Input constituent des éléments essentiels de l'interface utilisateur de notre application.

- **About** : Ce composant est utilisé pour fournir des informations sur l'application, un rappel du contexte du projet, la liste des fonctionnalités, et les mentions obligatoires.
- **Footer** : Le composant Footer est situé en bas de l'interface utilisateur et contient les noms des développeurs de l'application ainsi qu'une mention de copyright.
- **MenuBar** : La MenuBar est une barre de navigation présente en haut de toutes les pages de notre application.
- **Input** : Cette page est laissée vide et servira à Y. Asloudj pour décrire la structure des données plus en détail, dans la suite de sa recherche. C'était une demande de sa part.

5.3 Utils

Le répertoire *utils* contient des classes et fonctions qui sont utilisés dans les différents composants de l'application. Ces derniers ont été placés dans ce répertoire de façon à séparer les responsabilités efficacement entre les différents fichiers de notre projet pour permettre de plus facilement déboguer le code quand cela est nécessaire.

5.3.1 Color

Le fichier *Color.js* contient les fonctions permettant de choisir et fixer les couleurs des populations de cellules représentées dans le diagramme de Sankey.

Une couleur différente est choisie pour chaque population de cellules. Pour s'assurer que le contraste entre chaque couleur est suffisant, nous utilisons une fonction de la librairie D3 qui nous permet de générer une palette de couleurs. Cette dernière génère le nombre de couleurs requis, ayant une valeur RGB suffisamment éloignée les unes des autres pour être assez contrastées. De plus, nous savons grâce à notre client que les données importées sur l'application auront au maximum 20 populations de cellules distinctes. Après plusieurs tests d'utilisation de la palette, nous nous sommes rendu compte qu'une palette d'une vingtaine de couleurs permet une distinction facile des couleurs. Nous avons donc été confortés dans l'idée d'utiliser la palette de D3.

Une fois la palette de couleur créée, il nous a fallu attribuer une couleur à chaque population de cellules. Le client voulait qu'il y ait une cohérence dans les couleurs entre une population **parent** et ses populations **enfants**. Plus explicitement, le client voulait que les couleurs des populations enfants aient une teinte assez proche de celle de la population parent. Pour répondre à cette demande nous faisons un **parcours en profondeur** du diagramme de Sankey pour colorer nos populations, en avançant dans la palette de couleur qui suit le cercle des couleurs. De ce fait, le résultat n'est pas parfait et c'est normal, le problème COLORABILITÉ étant un des 21 problèmes NP-Complets de Karp.

Une fois les couleurs de chaque population sont définies, ils nous faut définir les couleurs des gènes présents dans ces dites populations. Pour cela, nous cherchons à colorier les gènes spécifiques à une population. Un gène spécifique à une population est un gène qui n'est exprimé que dans cette dernière ou dans ses populations enfants.

Ainsi, on vérifie pour chaque gène s'il n'est présent que dans une population et ses enfants, dans ce cas-là il sera de la couleur de la population parent. Si ce dernier

est présent dans plusieurs populations de sous-familles différentes, alors il n'est pas spécifique à une population et est donc affiché en gris. De ce fait, la sélection des couleurs permet de voir rapidement et facilement dans les histogrammes de quelles populations chaque gène est spécifique.

5.3.2 Constants

Lors du développement de l'application, nous avons remarqué que nous avions dans notre code un certain nombre de constantes et de *magic numbers* telles que la taille du diagramme, la largeur des nœuds, etc. Nous avons donc convenu, après échange avec notre client, qu'un fichier regroupant ces constantes serait judicieux pour lui permettre de modifier, à sa guise ces valeurs, sans avoir à les chercher directement dans le code. Nous avons donc sélectionné les variables qui nous ont semblé être les plus judicieuses à pouvoir être éditées par le client et nous les avons déplacées. Cela rend ainsi le code plus modulable et plus compréhensible grâce aux noms explicites des constantes utilisées. Ce fichier répond ainsi au besoin du client de pouvoir, à l'avenir, modifier et améliorer l'application que nous lui livrons.

5.3.3 SankeyStructure

Au début du développement de ce projet, nous passions directement en paramètre du composant Sankey, les données du fichier importé. Cependant, lorsque nous avons commencé à implémenter les couleurs, nous nous sommes rendu compte qu'une structure de données arborescente serait bien plus appropriée pour représenter le diagramme de Sankey.

La classe SankeyStructure définit et construit, à partir du jeu de donnée importé, la structure arborescente correspondant. Nous pouvons ainsi facilement obtenir différentes informations sur notre arbre telles que les relations parent-enfants de chaque population.

Cette classe permet ainsi de bien séparer les responsabilités de chaque fichier. Le composant du diagramme de Sankey n'a ainsi pas à gérer le traitement des données importées vers la construction du diagramme. La construction de la structure de données du diagramme étant faite dans son fichier propre : SankeyStructure.

5.3.4 Validation

Les tests de validité des données effectués lors de l'import du fichier sont définis dans le fichier *Validation.js*. Ces derniers ont été décidé après que notre client nous ai clarifié et confirmé le format définitif des fichiers que notre application doit pouvoir traiter. Ces tests sont les suivants :

- Vérifier que les données importées contiennent bien 2 tableaux et que ces derniers se nomment *meta* et *markers*.
- Vérifier qu'une seule population de cellule est la racine, c'est-à-dire qu'elle n'a pas de parents.
- Vérifier que la valeur de consensus dans le tableau *meta* est bien un F1 Score compris dans l'intervalle $[0, 1]$.
- Vérifier que toutes les populations, exceptée la racine, ont bien un parent (dans le tableau *meta*).
- Vérifier qu'il n'y a pas de circularité dans les relations parent-enfants.
- Vérifier que toutes les populations de cellules ont des valeurs *n* et *consensus* définies et valides.
- Vérifier que chaque population présente dans le tableaux *meta* est aussi présente dans le tableau *markers*.
- Vérifier que chaque population présente dans le tableaux *markers* est aussi présente dans le tableau *meta*.
- Vérifier que les valeurs dans le tableau *markers* sont bien positives.

Si le fichier importé passe avec succès toutes ces vérifications, alors les données peuvent être utilisées pour générer une instance de la classe *SankeyStructure*, pour ensuite créer un diagramme de Sankey à l'aide la librairie D3.

Cependant, si certaines vérifications ne passent pas, des messages d'erreurs sont générés et sont alors affichés sous forme de *toasts*, ou fenêtres contextuelles, contenant la liste des erreurs correspondant au fichier importé. Cela permet à l'utilisateur de comprendre facilement **toutes** les raisons pour lesquelles le fichier n'a pas pu être chargé dans l'application.

5.4 Diagrammes

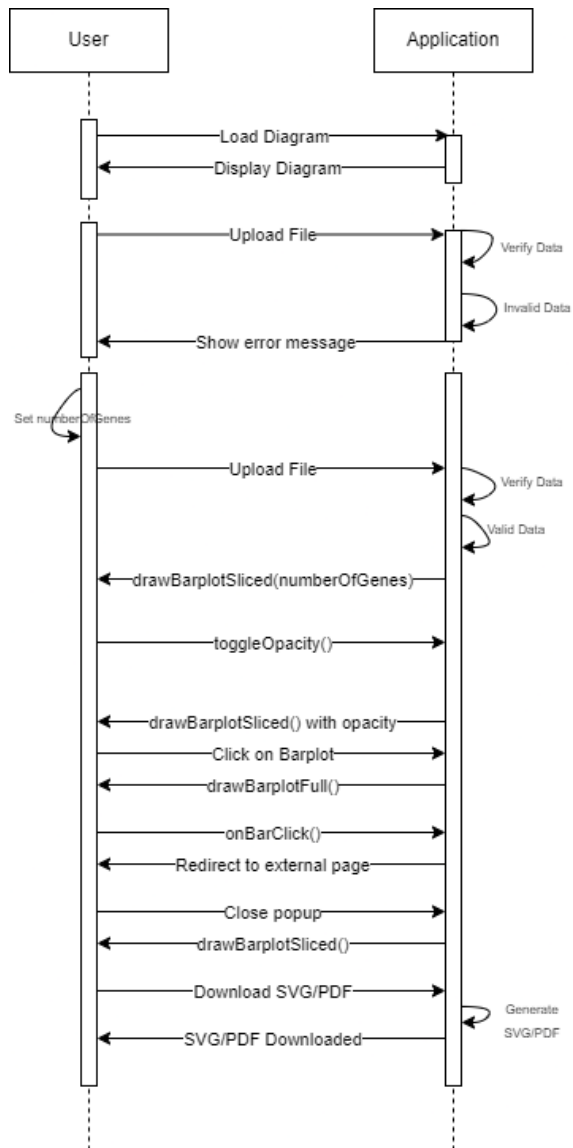


FIGURE 9 – Diagramme de séquence de l'application

6 Tests et tactiques

Dans cette section, nous tenterons de vous expliquer comment nous avons écrit nos tests et quelles ont été nos stratégies.

6.1 Tests unitaires

Comme rappelé précédemment, nous utilisons *Jest* et *React Testing Library* [12]. Le premier fournit une API intuitive pour écrire des tests et exécuter des assertions. Il peut exécuter des tests en parallèle, ce qui accélère grandement le processus de test. *React Testing Library* est quant à elle très utile pour tester des composants React en **simulant** le comportement de l'utilisateur.

Ils ont généralement la même structure à chaque fois, que l'on peut constater grâce à l'extrait ci-dessous :

```
1 import React from 'react';
2 import { render, screen, fireEvent, waitFor } from '@testing-library/react';
3 import MyComponent from '../components/MyComponent';
4
5 describe('MyComponent', () => {
6     const mockData = null; // You can mock data structures or variables
7
8     it('My first test', () => {
9         render(MyComponent test);
10
11         // Testing code
12         { ... }
13
14         // Assertions
15         expect(true); // or false, String, Number, any ...
16     });
17 });
```

FIGURE 10 – Squelette d'un code de test React typique

L'intégralité des tests est répertoriée dans les pipelines **GitLab**. Pour les consulter, il faut y accéder en passant par l'onglet *Build* dans la section *Pipelines*. Dès que vous êtes dans ce menu, accédez simplement au dernier commit réussi et cliquez dessus.

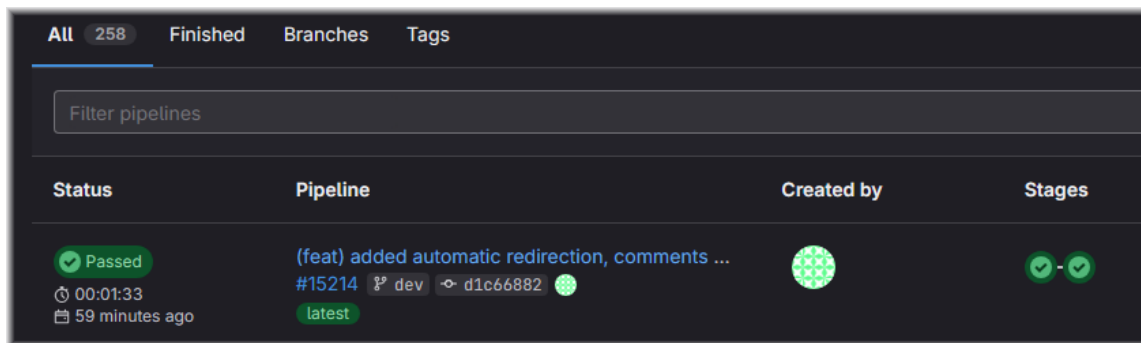


FIGURE 11 – Capture d’écran du menu listant tous les pipelines du dépôt

Il suffit ensuite d’accéder à l’onglet *Tests* qui nous répertorie tous les tests exécutés pour ce commit précis.

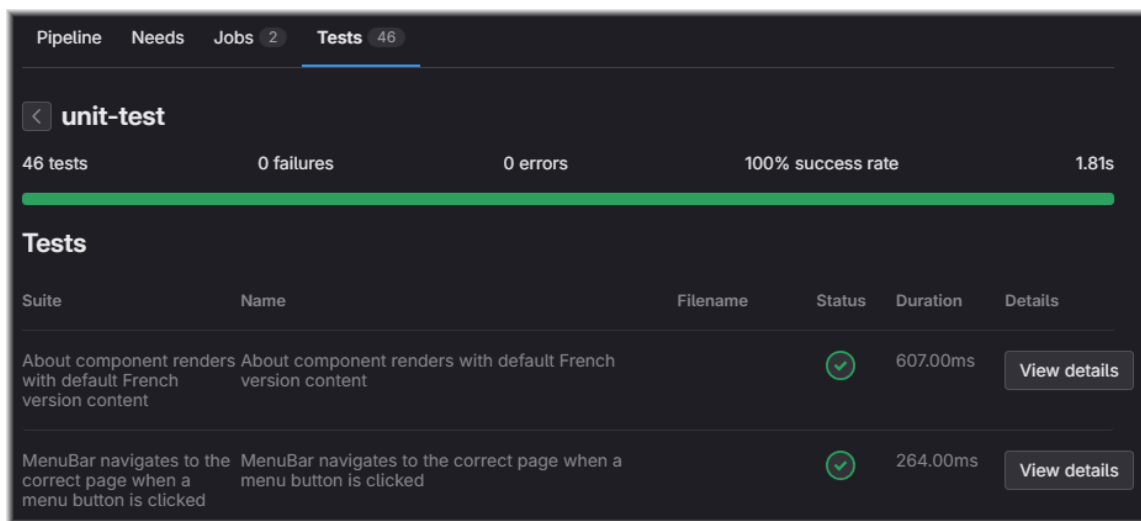


FIGURE 12 – Capture d’écran du résumé de tous les tests d’un commit

6.1.1 Tests d’acceptation

Les tests d’acceptation ou tests positifs vérifient le bon fonctionnement de l’application. Pour ces tests positifs, nous avons écrit un large nombre de *happy path tests*, c’est-à-dire des scénarios où l’on attend que l’application réagisse comme prévu sans rencontrer d’erreurs, d’exceptions ni de comportements inattendus.

Dans la suite du rapport, nous allons prendre comme exemple les tests pour vérifier et valider les données *uploadées* par l’utilisateur, car leurs scénarios sont les plus faciles à comprendre.

```
1 import { render, screen } from '@testing-library/react';
2 import { FileImport } from '../components/FileImport.js';
3 import { BrowserRouter as Router } from 'react-router-dom';
4 import { TOAST_MESSAGES, Validation } from '../utils/Validation.js';
5
6 describe('validation', () => {
7   it('returns correct result for valid data', () => {
8     // This data is valid, there's only one root
9     // There is all the required values and they
10    // are within their interval
11    const data = new Map();
12
13    data.set("meta", [{ "": "cell1", "parent": null, "n": 10, "consensus": 0.15 }]);
14    data.set("markers", [{ "": "cell1", "marker1": 5, "marker2": 8 }]);
15
16    const result = Validation(data);
17    expect(result[0]).toBe(true);
18  });
19 });
```

FIGURE 13 – Extrait d’un scénario de test positif du fichier `FileImport.test.js`

6.1.2 Tests négatifs

Les tests négatifs vérifient le comportement du logiciel lorsqu'il est soumis à des conditions incorrectes ou invalides. En quelque sorte, il s'agit de tester le complémentaire du domaine valide, de ce qu'on teste avec les tests d'acceptation. Voici plusieurs exemples sur la vérification et validation des données ci-dessous :

```
1 describe('validation', () => {
2     // Skipping beginning of the describe() code block ...
3
4     it("returns incorrect result for a cell with non-respectful 'consensus' value",
5     () => {
6         const data = new Map();
7         data.set("meta", [
8             { "": "cell1", "parent": null, "n": 10, "consensus": 0.33 },
9             { "": "cell2", "parent": "cell1", "n": 10, "consensus": 1.01 }]);
10        data.set("markers", [
11            { "": "cell1", "marker1": 5, "marker2": 4 },
12            { "": "cell2", "marker1": 5, "marker2": 8 }]);
13
14        const result = Validation(data);
15        expect(result[0]).toBe(false); // 1.01 is wrong
16        expect(result[1]).toContain(TOAST_MESSAGES["CONS_ERROR"]);
17    });
18
19    it("returns incorrect result for a cell with negative value in the 'markers' sheet",
20    () => {
21        const data = new Map();
22        data.set("meta", [
23            { "": "cell1", "parent": null, "n": 10, "consensus": 0.33 },
24            { "": "cell2", "parent": "cell1", "n": 10, "consensus": 0.55 }]);
25        data.set("markers", [
26            { "": "cell1", "marker1": 5, "marker2": -4 },
27            { "": "cell2", "marker1": 5, "marker2": 8 }]);
28
29        const result = Validation(data);
30        expect(result[0]).toBe(false); // -4 is wrong
31        expect(result[1]).toContain(TOAST_MESSAGES["NEGATIVE_ERROR"]);
32    });
33 });
```

FIGURE 14 – Extrait de scénarios de tests négatifs du fichier `FileImport.test.js`

6.2 Tests de domaine

6.2.1 Tests aux limites

Les tests aux limites vérifient le comportement par rapport à des entrées se situant sur les frontières des domaines de l'application. En général, il s'agit de simuler une activité bien supérieure à l'activité normale, pour voir comment le système réagit aux limites du modèle d'usage de l'application. Ces tests, aussi appelés *boundary value tests* en anglais, vérifient les comportements aux limites du (*non*)-fonctionnement.

Pour cela, nous utilisons un assez gros jeu de données fourni par notre client, *Baron.xlsx* que vous pourrez également retrouver dans le dossier *data* de notre projet. Ce jeu de données contient 37 populations de cellules et 3 557 gènes au total tandis que notre jeu de données principal, *Darmanis.xlsx* contient quant à lui 15 populations de cellules et 1 338 gènes.

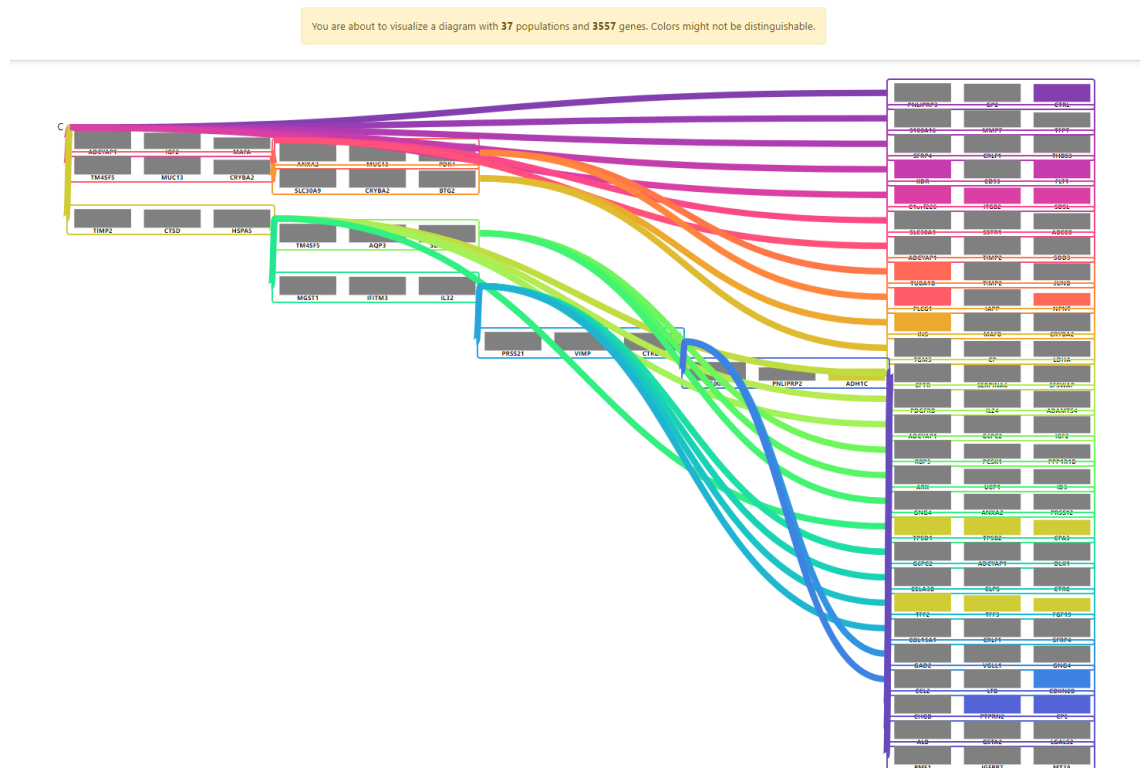


FIGURE 15 – Capture d'écran du diagramme généré depuis le fichier *Baron.xlsx*

Comme vous pouvez le remarquer, le diagramme de Sankey se génère bien mais l’affichage de ce dernier n’est pas si lisible. Un message d’alerte est tout de même affiché si le nombre de populations de cellules est supérieur à 20 et si le nombre de gènes total est supérieur à 2000. Ce sont les limites que nous avons fixé à partir desquelles le rendu n’est plus aussi bon et où les couleurs peuvent ne sont pas forcément distinguables des unes des autres.

Nous sommes conscient que ce type de tests aux limites peut rentrer dans la catégorie des tests unitaires, mais après maintes discussions avec notre client, notre philosophie reste d’accepter tous les fichiers valides, mais d’avertir l’utilisateur des potentielles répercussions sur son expérience.

7 Analyse des sprints effectués

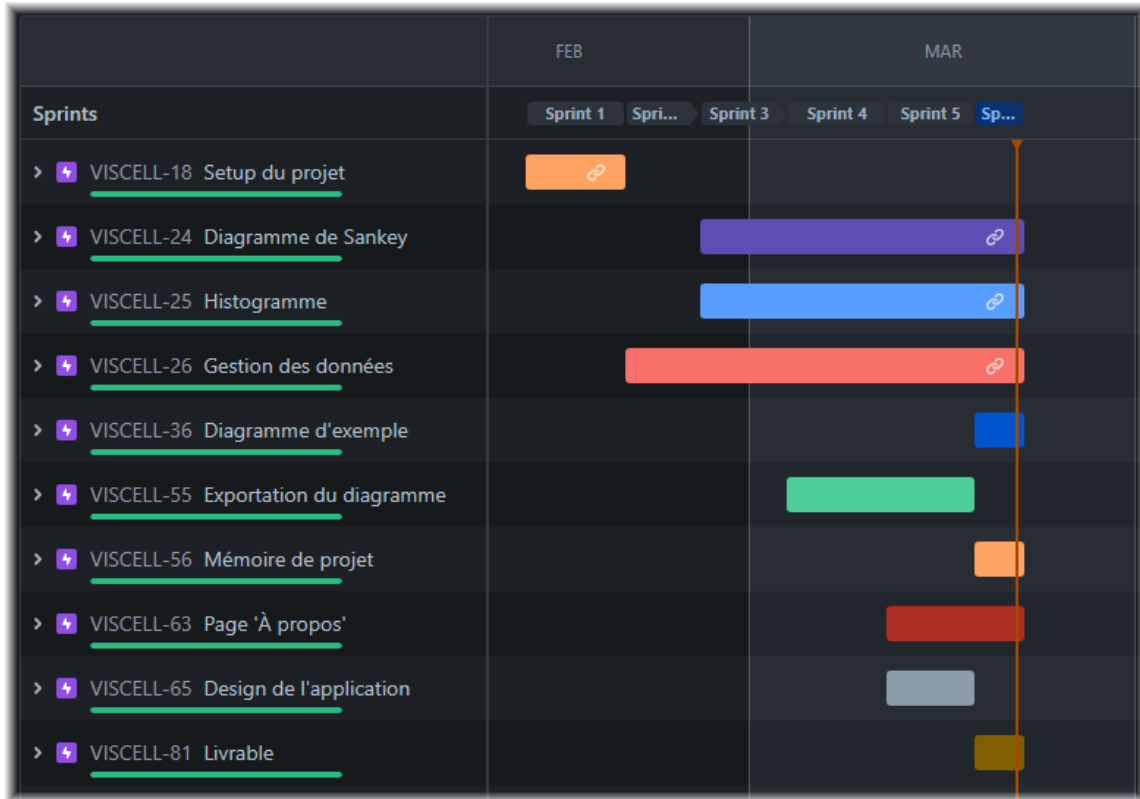


FIGURE 16 – Chronologie des sprints

Lors de ce projet, nous devons suivre la méthode Scrum pour la gestion de ce dernier. Ainsi, nous organisons notre développement de façon agile en découpant nos phases en plusieurs **sprints**. Un sprint est une période de temps pendant laquelle nous avons à réaliser une quantité de travail que nous nous fixons en amont.

Ce projet, d'une durée totale de 40 jours environ, a été divisé en 6 Sprint d'une semaine chacun. Chaque semaine, après la réunion d'avancement du projet avec notre client, nous définissions les tâches à effectuer pour la semaine à venir.

Pour évaluer nos performances lors de ces sprints, nous pouvons observer les *burndown charts* associés. Ces derniers représentent la quantité de travail restante d'un sprint à un temps donné. Plus ce graphique est constant, plus la performance du groupe est en adéquation avec les prévisions de ce dernier.

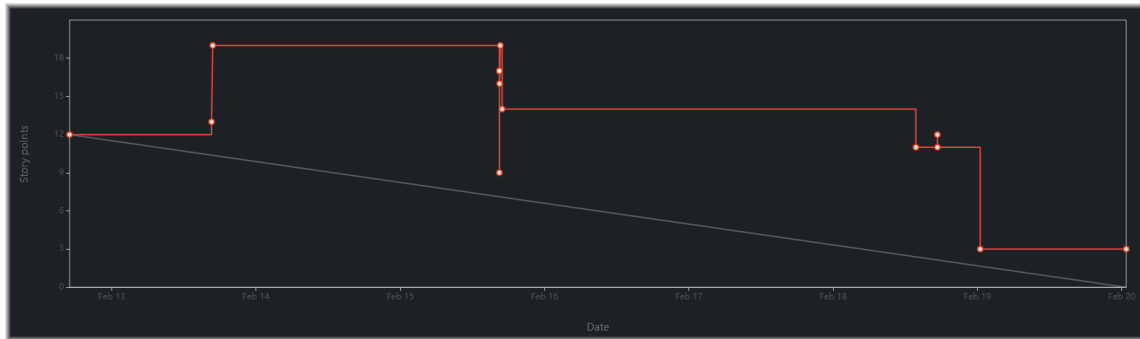


FIGURE 17 – Rapport de burndown du premier Sprint

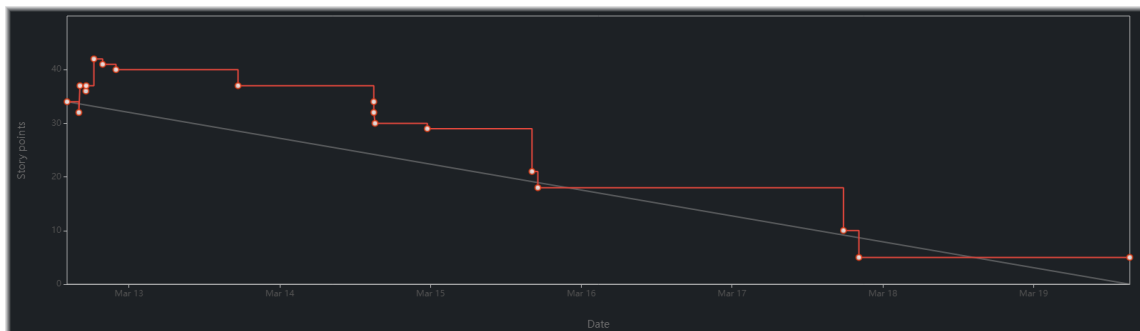


FIGURE 18 – Rapport de burndown du cinquième Sprint

Ainsi si l'on observe ces graphiques, on peut se rendre compte que les premiers sprints n'étaient pas réalisés de façon linéaire. En effet, beaucoup de tâches étaient finalisées dans les derniers jours, ce qui créait une dette technique qui se répercutait sur le sprint suivant. Cependant, au fur et à mesure de l'avancée du projet, nos performances lors des sprints sont devenues plus constantes et en adéquation avec nos prévisions.

8 Utilisation d'IA génératives

Au cours du développement du projet, nous avons été amenés à utiliser des outils d'IA génératives pour nous faciliter certaines tâches redondantes et répétitives. Il est important pour nous de ne pas en devenir dépendant et d'être critique vis-à-vis de ces technologies et des abus qui peuvent en découler.

Nous avons utilisé trois IA génératives relativement populaires dans notre domaine, **ChatGPT**, **GitHub Copilot** et **Phind**.

Ces technologies sont aujourd'hui omniprésentes dans notre domaine, il est inutile de faire comme si elles n'existaient pas. Cependant, leur utilisation soulève également des questions éthiques et des préoccupations concernant la propriété intellectuelle, la qualité du code généré et la dépendance excessive à l'égard des machines. Il est donc essentiel de les utiliser de manière réfléchie et responsable, en comprenant à la fois leurs capacités et leurs limites, tout en restant vigilants quant à leurs implications à long terme.

Les utilités de ces outils pour notre projet ont porté sur :

- La recherche de syntaxe spécifique à un langage
- La complétion rapide de certaines lignes de code
- L'optimisation algorithmique
- La traduction
- La suppression de code dupliqué

Le gain de temps et d'énergie généré par l'utilisation de ces outils est une plus value non négligeable pour des phases de développement comme celles-ci. Il nous est ainsi important de citer ces outils de développement qui ont contribué au bon déroulement du projet.

9 Améliorations Possibles

Nous avons réalisé une application complète en accord avec les souhaits de notre client. Cependant, au cours du développement nous avons pu noter plusieurs idées que nous aurions implémenté sans contrainte de temps :

- **Affichage sous forme verticale** : En voyant le diagramme de Sankey mis en place, semblable à un arbre, il nous a semblé pertinent de pouvoir le représenter verticalement avec la racine à son sommet. Malheureusement, après un temps de recherche et une exploration fastidieuse des différents *forks* de la librairie D3, cette tâche s’est avérée plus ardue en raison des librairies utilisées, et a donc été repoussée.
- **Choix de palette de couleurs** : Pendant la mise en place de l’algorithme de coloration du graphe il nous est venu à l’esprit, via les ressources mises en place par D3, de pouvoir éventuellement choisir la palette de couleur utilisée. Cette fonctionnalité est resté au stade d’idée tant elle nous semblait moins utile en comparaison du reste.
- **Tutoriel interactif** : De part la complexité des données traitées, il nous est venu l’idée avec notre client, d’imaginer un tutoriel interactif à l’aide de bulles de texte par exemple. De cette manière, la signification des valeurs, couleurs, et forme des diagrammes serait accessible à n’importe qui. Étant donné que nous n’allions pas déployer notre application, cette étape nous a paru prématurée et peu pertinente à ce stade du développement.
- **Prévisualisation des hyperliens** : Au cours d’une réunion hebdomadaire avec notre client, il a été évoqué une prévisualisation des hyperliens à même le site, en survolant un gène. Nous avons préféré repousser cette tâche d’une part à cause de sa complexité mais aussi à cause de la piètre qualité d’affichage du site répertoriant tous les gènes.

10 Conclusion

Le bilan de ce projet est positif, tant pour notre équipe de développement que notre client. Nous avons su répondre à ses attentes et remplir tous les objectifs que nous nous étions fixé.

Nous pouvons constater que des améliorations du logiciel ont été proposées indépendamment du travail qui était attendu. De part l'ampleur du projet, nous avons chacun pu consolider nos compétences sur des outils de développement logiciel majeurs, mais également apprendre et découvrir de nouvelles technologies, et méthodes de travail.

La mise en place de la méthode **Scrum** et l'utilisation des **Sprints** nous a demandé une organisation solide pour répartir les tâches pour une équipe de 6 personnes. Cela nous a aussi demandé d'acquérir des compétences autres que celles d'un développeur.

A Maquettes

Très rapidement dans le projet, nous avons réalisé des maquettes pour s'assurer de partir dans la bonne direction et fournir rapidement un produit sur lequel discuter avec notre client. L'application a beaucoup évolué depuis la réalisation de ces maquettes, ces dernières représentant non pas un objectif final mais bien un objectif à court terme et récapitulant les besoins nécessaires du client.

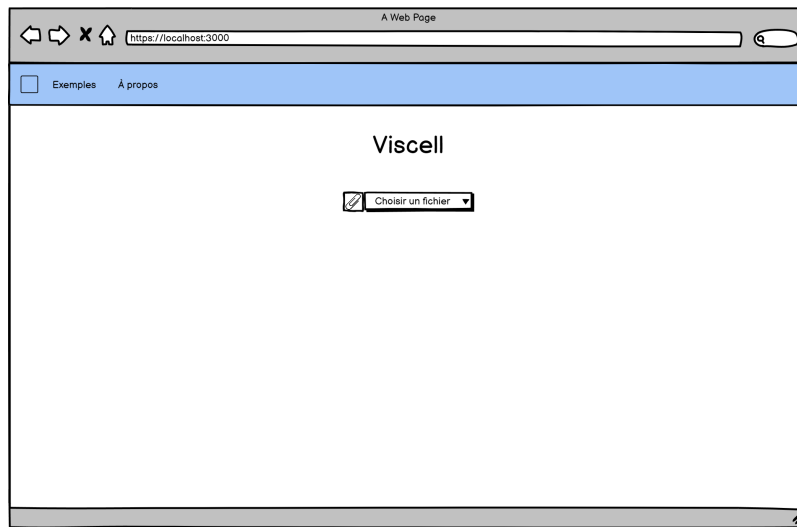


FIGURE 19 – Page d'accueil imaginée pour la maquette

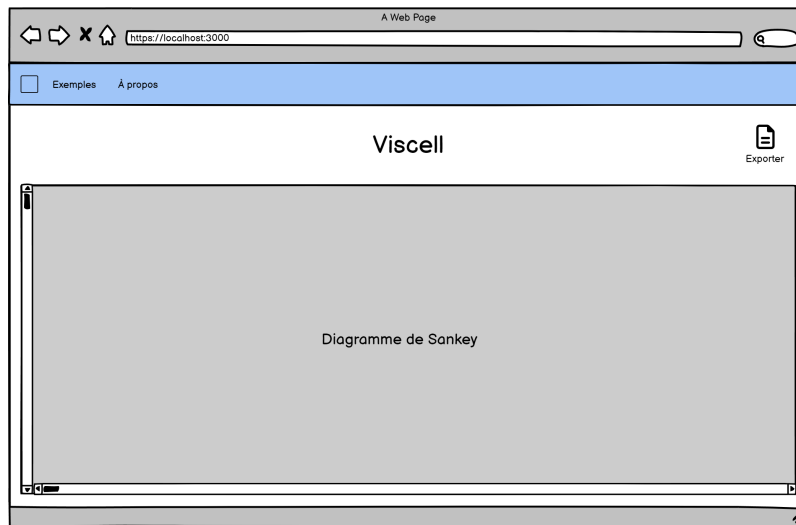


FIGURE 20 – Page d’accueil une fois le fichier soumis

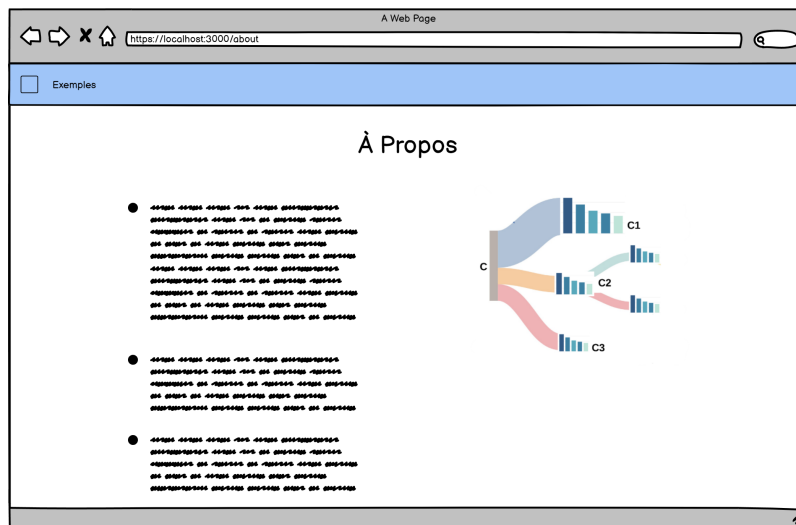


FIGURE 21 – Page À propos

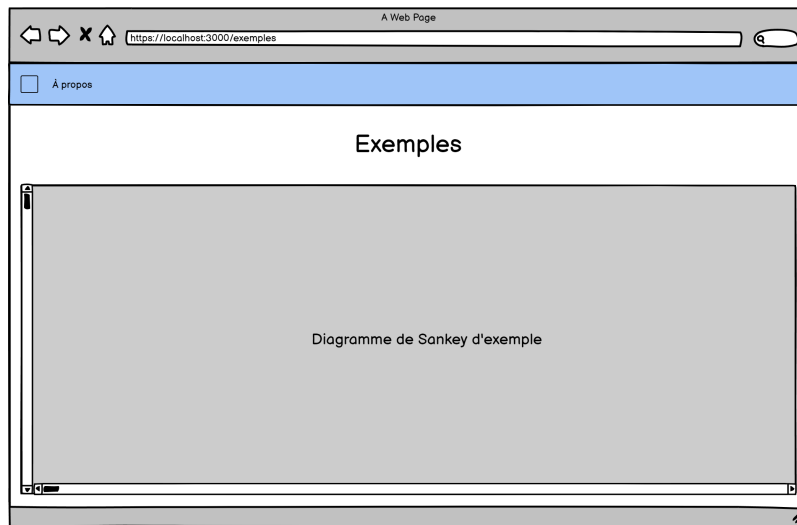


FIGURE 22 – Page d'accueil avec le diagramme d'exemple

Références

- [1] WIKIPEDIA. *Sankey diagram*. https://en.wikipedia.org/wiki/Sankey_diagram.
- [2] WIKIPEDIA. *F-score*. <https://en.wikipedia.org/wiki/F-score>.
- [3] PUBCHEM. *United States National Library of Medicine (NLM)*. <https://pubchem.ncbi.nlm.nih.gov/>.
- [4] REACT GOOGLE CHARTS. *Sankey*. <https://www.react-google-charts.com/examples/sankey>.
- [5] PLOTLY. *Sankey Diagrams in JavaScript*. <https://plotly.com/javascript/sankey-diagram/>.
- [6] D3.JS. *Sankey Diagram*. <https://d3-graph-gallery.com/sankey.html>.
- [7] META. *React*. <https://react.dev/>.
- [8] ATlassian JIRA. *Feature Branch Workflow*. <https://www.atlassian.com/git/tutorials/comparing-workflows/feature-branch-workflow>.
- [9] JEST. *Jest · Delightful JavaScript Testing*. <https://jestjs.io/>.
- [10] DARMANIS, S., SLOAN, S. A., ZHANG, Y., ENGE, M., CANEDA, C., SHUER, L. M., HAYDEN GEPHART, M. G., BARRES, B. A., AND QUAKE, S. R. « A survey of human brain transcriptome diversity at the single cell level. » In : *Proceedings of the National Academy of Sciences* 112.23 (2015), p. 7285-7290. DOI : [10.1073/pnas.1507125112](https://doi.org/10.1073/pnas.1507125112). arXiv : [2002.05651](https://arxiv.org/abs/2002.05651) [cs.CY].
- [11] BARON, M., VERES, A., WOLOCK, S. L., FAUST, A. L., GAUJOUX, R., VETERE, A., RYU, J. H., WAGNER, B. K., SHEN-ORR, S. S., KLEIN, A. M., MELTON, D. A., AND YANAI, I. « A Single-Cell Transcriptomic Map of the Human and Mouse Pancreas Reveals Inter- and Intra-cell Population Structure. » In : *Cell systems* (2016), p. 346-360. DOI : [10.1016/j.cels.2016.08.011](https://doi.org/10.1016/j.cels.2016.08.011).
- [12] TESTING LIBRARY. *React Testing Library*. <https://testing-library.com/docs/react-testing-library/intro/>.