programmez!
N°270
JUILLET
AOUT
2025

Le magazine des dévs - CTO & Tech Lead

UN ÉTÉ QUANTIQUE

Emulation, programmation, algos, sécurité, hardware

Cloud français partie 3

avec clever cloud

Ada : un langage qui mérite d'être (re)connu !

Les nouveautés de Python 3.13

Eco-conception : des entreprises témoignent 10 bonnes pratiques pour ton site web

Qubits fournis par EVIDEN

Printed in EU - Imprimé en UE - BELGIQUE 7,50 € P928345 - Canada 10,55 $ CAN - SUISSE 14,10 FS - DOM Surf 8,10 € - TOM 1100 XPF - MAROC 59 DH

M 04319 - 270 - F: 6,99 € - RD

PORTAGE

Fiparco Portage

Portage salarial

Votre carrière est une priorité !

Votre contact

Ikram 07 86 45 01 75

fiparco-portage.fr

FIPARCO Sponsor de PROGRAMMEZ !

SOMMAIRE 270

Divers

EDITO INVITÉ
CHRISTOPHE PRUGNAUD #270

La souveraineté numérique est-elle accessoire ?

Les événements technologiques de ces derniers mois nous apportent un point de vue éclairant sur ce vaste sujet.

- En 2024, un fournisseur historique de solution de virtualisation a décidé de façon unilatérale à procéder à l'augmentation significative de ses tarifs.
- Depuis le début de l'année, les taxes douanières US fluctuent chaque jour dans le sillage de la politique de leur nouveau président élu.
- Nos alliés européens prennent conscience que leurs avions F-35 pourraient ne pas être pleinement opérationnels sans l'aval des Américains.
- Plus récemment, le procureur général de la Cour pénale internationale a perdu l'accès à ses emails.
- Et depuis fin mai nos photos, vidéos et commentaires sur certaines plateformes de communication servent à l'entraînement d'une IA.

Vous voyez poindre le rapport entre des modifications de coûts, des arrêts sans préavis de services, des entraves au bon fonctionnement ou l'accès à vos données : la souveraineté numérique.

Un petit rappel légal s'impose. La souveraineté est le fait qu'un État ne soit soumis à aucun autre État.

En France, la souveraineté est inscrite dans les articles 1 et 3 de la constitution de 1958 : « Le Peuple français proclame solennellement son attachement aux Droits de l'Homme et aux principes de la souveraineté nationale … La souveraineté nationale appartient au peuple qui l'exerce par ses représentants et par la voie du référendum. Aucune section du peuple ni aucun individu ne peut s'en attribuer l'exercice. »

Il convient de ramener cette définition très formelle à nos actions du quotidien dans l'IT. Chaque jour, nous créons de nouveaux logiciels, nous enrichissons fonctionnellement nos solutions IT, nous déployons des infrastructures et nous collectons et restituons de nombreux pétaoctets de données pour nos services numériques.

Et dans cette extraordinaire dynamique technologique française, nous constatons que de nombreux acteurs étrangers se trouvent en capacité d'interrompre un service ou d'accéder aux données en dehors du champ d'application de la souveraineté, presque en toute légitimité de leur point de vue.

Un sous-traitant usant de pratiques contractuelles déloyales va mettre à mal un modèle économique qui était jusque-là viable. La fermeture d'un compte de service dans le cloud peut stopper un ensemble de services essentiels de l'état. Les données collectées peuvent être consultées sur la simple demande d'un juge étranger respectant simplement le Cloud Act et la section 702 du FISA. L'usage d'une extension de nom de domaine internet peut être limité par son exploitant de premier niveau (TLD). Une transaction en dollar US permet à un juge de faire le rattachement du service final opéré au droit américain. Et il existe des possibilités d'ingérences dans toutes les couches IT d'une solution : le DNS, internet, l'énergie, les télécommunications, le matériel, le stockage, la virtualisation, les middlewares …

Pour autant, notre définition de la souveraineté est implacable, le peuple par le peuple, est le seul à pouvoir stopper un service ou accéder aux données dans le cadre législatif. Ce droit s'exerce par la décision d'un juge faisant respecter une loi votée par les représentants du peuple.

Et c'est là que nous devons entrer dans l'action. Pour réduire tous ces risques et remettre notre souveraineté au centre du débat, nous devons collectivement façonner le monde numérique de demain mais aussi ses sous-jacents techniques et matériels.

Il nous faut gérer et administrer nos propres datacenters. Nous devons développer et déployer nos solutions de cloud computing. La contribution massive aux projets opensource et la création de brique d'infrastructure doivent être une priorité. Et enfin, utilisons tout ce qui existe déjà dans le riche écosystème IT français et européen.

La géopolitique de notre époque, bien que basée sur des amitiés et des alliances tissées de longue date, souffre des maux de l'hyperconnectivité du monde moderne entrant en opposition avec le renfermement de chacun sur soi. Ne pas mettre la souveraineté en œuvre dans l'IT nous exposera inévitablement à la fuite de nos données, à l'ingérences dans nos affaires intérieures voire à un shutdown technologique.

La souveraineté n'est donc pas accessoire, elle est au cœur de nos préoccupations. Dans l'esprit des dossiers Cloud français publiés ces derniers mois dans votre revue, il est évident que nous devons collectivement renforcer notre paysage IT français et européen et faire en sorte qu'il soit sécurisé et résilient.

programmez!

Directives de compilation

Programmez! n°270
JUILLET - AOUT 2025

Directeur de la rédaction : **Jean-Christophe Tic**
Rédacteur en chef : **François Tonic**
ftonic@programmez.com

Contacter la rédaction
redaction@programmez.com

Les experts techniques du numéro

Christophe Prugnaud
Quentin Adam
Pierre Zemb
Alexandru Lata
Raphaël Nicoud
Stéphane Carrez
Damien Nicolazic
Grégory Vieux
Mariem Bahri
Olivier Hess
Robert Wang
Ali Abbassi
Yann Dujardin
Philippe Lacombe
Caroline Prodhon
Simon Fried
Eric Bourreau
Gérard Fleury
Martin Bombardelli
Marc Sevaux
Samuel Horsch
Jean-Michel Torres
Benjamin Gigon
Raphaël Semeteys
Haroun Azoulay
Clément Sannier
Mathieu Cottard
Benoît Prieur
Lamine Bensaid
Jaideep Varier
Jean-Pierre Roullin
François Lankar
Frédéric Bordage

Maquette
Pierre Sandré

Marketing – promotion des ventes
Agence BOCONSEIL - Analyse Media Etude
Directeur : Otto BORSCHA
oborscha@boconseilame.fr
Responsable titre : Anthony Carrée
Téléphone : 09 67 32 09 34

Publicité
Nefer-IT
Tél. : 09 86 73 61 08
ftonic@programmez.com

Impression
Léonce Deprez, France

Dépôt légal
A parution

Commission paritaire
1225K78366

ISSN
1627-0908

Abonnement
Abonnement (tarifs France) : 55 € pour 1 an, 90 € pour 2 ans. Etudiants : 45 €. Europe et Suisse : 65 €
Algérie, Maroc, Tunisie : 64 € - Canada : 80 €
Tom - Dom : voir www.programmez.com.
Autres pays : consultez les tarifs sur www.programmez.com.
Pour toute question sur l'abonnement :
abonnements@programmez.com

Abonnement PDF
monde entier : 45 € pour 1 an.
Accès aux archives : 25 € (1 an)
30 € (2 ans).

Nefer-IT
150, rue Lamarck, 75018 Paris
redaction@programmez.com
Tél. : 09 86 73 61 08

Abonnez-vous à programmez!

Formules	1 an	2 ans	Etudiant	Numérique
Vous recevez le magazine chez vous	1 an 10 numéros (papier) 55 €	2 ans 20 numéros (papier) 90 €	1 an 10 numéros (papier) 45 €	1 an 10 numéros (format PDF) 45 €
Abonnements + accès aux archives	80 € (1 an + archives)	120 € (2 ans + archives)		70 € (1 an + archives)

Tarifs France métropolitaine.
Tarif PDF : valable partout dans le monde

L'abonnement comprend : les numéros normaux et les hors-séries

L'abonnement **numérique** est disponible uniquement sur notre boutique : **www.programmez.com**

Oui, je m'abonne

ABONNEMENT à retourner avec votre règlement à :
PROGRAMMEZ, Service Abonnements
57 Rue de Gisors, 95300 Pontoise

☐ Abonnement 1 an :	55 €
☐ Abonnement 2 ans :	90 €
☐ Abonnement 1 an Etudiant : Photocopie de la carte d'étudiant à joindre	45 €

Option : accès aux archives

☐ Pour abonnement 1 an	25 €
☐ Pour abonnement 2 ans	30 €

☐ Mme ☐ M. Entreprise : Fonction :

Prénom : Nom :

Adresse :

Code postal : Ville :

Adresse email indispensable pour la gestion de votre abonnement

E-mail : @

☐ Je joins mon règlement par chèque à l'ordre de Programmez !

☐ Je souhaite régler à réception de facture

* Tarifs France métropolitaine

Offres pouvant s'arrêter à tout moment, sans préavis **PROG HS19**

conférences programmez!

DevCon #25 : informatique quantique

jeudi 9 octobre

Au campus 42 Paris - Accueil à partir de 13h30.
Début des sessions : 14h
Appel à session : https://forms.gle/84rZrpCUQqpW73iM7

DevCon #26 : sécurité & hacking édition 2025

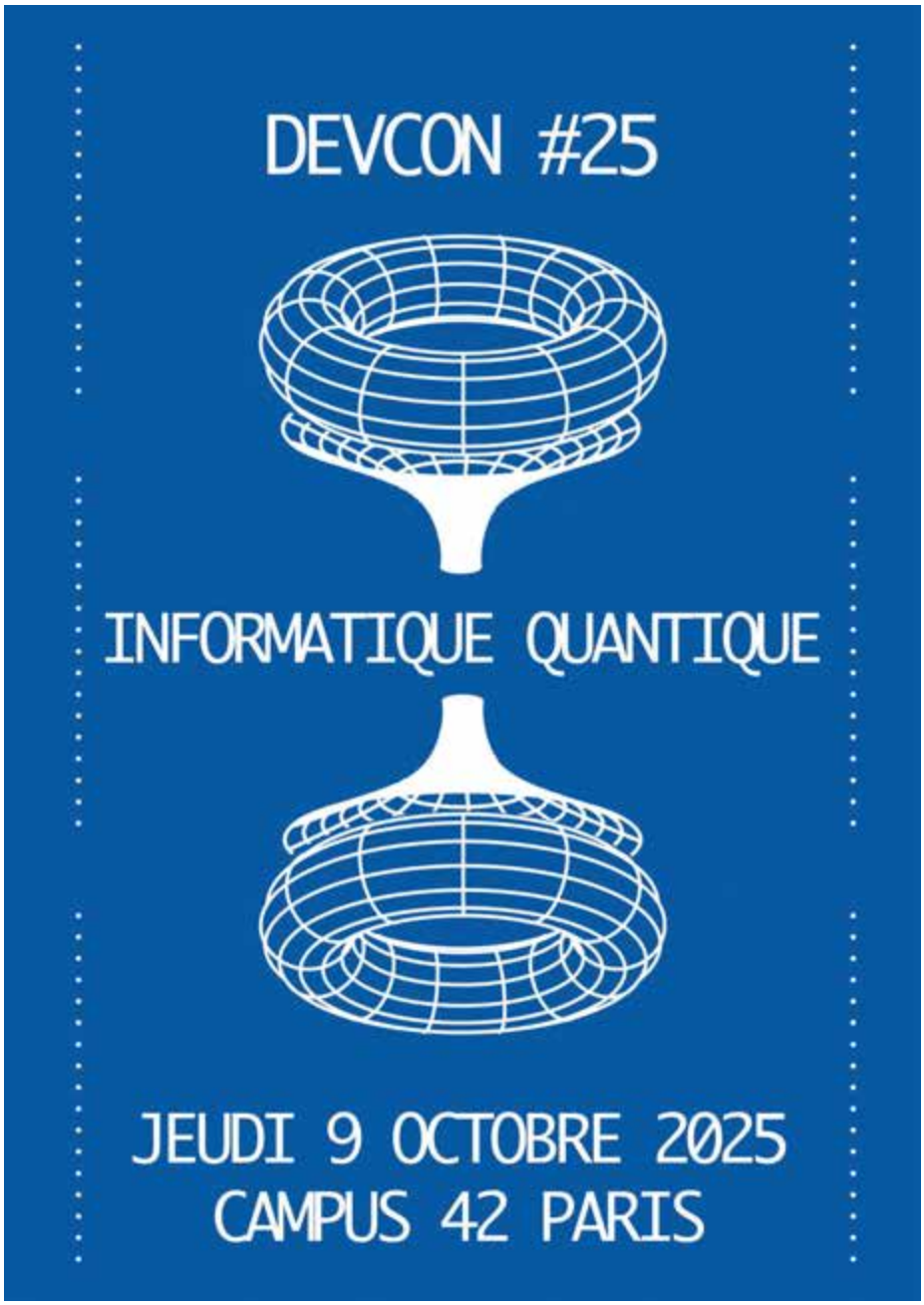
décembre

Nos prochains meetups :

30 septembre
4 novembre
16 décembre

WeScale (Paris) nous accueillera :
34 Bd Haussmann, 75009 Paris

informations & inscription : programmez.com



JUILLET 2025

Lun.	Mar.	Mer.	jeu.	Ven.	Sam.	Dim.
	1	2	3	4	5	6
7	8	9	10	11	12	13
Riviera Dev / Sophia Antipolis						
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30	31			

Merci à Aurélie Vache pour la liste 2025, consultable sur son GitHub :
https://developers.events/#/2025/calendar

PROCHAINS NUMÉROS

PROGRAMMEZ! N°271
Disponible à partir du
5 septembre 2025

HORS SÉRIE N°19
Disponible depuis le
27 juin 2025

programmez!

Le magazine des dévs - CTO & Tech Lead

www.programmez.com

SPÉCIAL ÉTÉ 2025
HORS-SÉRIE #19

GREEN-IT ECO-CONCEPTION

Langages, infrastructures & serveurs, sites web : des développements plus économes sont possibles

En partenariat avec

Printed in EU - Imprimé en UE - BELGIQUE 7,50 € P928345 - Canada 10,55 $ CAN - SUISSE 14,10 FS - DOM Surf 8,10 € - TOM 1100 XPF - MAROC 59 DH

M 01642 - 19 - F: 6,99 € - RD

Disponible depuis le 27 juin 2025

Cloud français PARTIE 3

Quentin Adam
CEO de Clever Cloud

clever cloud

Développeurs : votre rôle clé pour relancer l'économie française

Je pense que l'on sera tous d'accord sur ce point : **la France traverse une période économique difficile et incertaine.** Depuis plusieurs décennies, notre base industrielle s'est effritée. Les chaînes de production ont été éclatées à l'échelle mondiale, la valeur s'est déplacée, les marges se sont réduites. Nous avons conservé une économie de services solide, mais **nous avons perdu notre capacité à produire de la richesse structurante, récurrente, maîtrisée.**

Le numérique, un secteur pas comme les autres

Cette faiblesse structurelle est d'autant plus problématique que **le capital ne se comporte pas de la même manière dans tous les secteurs**. L'économie industrielle classique, celle qui constitue encore l'essentiel du CAC 40, repose sur un **rendement décroissant du capital** : plus on produit, plus les coûts marginaux augmentent, plus la rentabilité se tasse.

À l'inverse, **le numérique repose sur un rendement croissant du capital**. Une fois le logiciel développé, chaque nouvel utilisateur coûte presque zéro, et augmente pourtant la valeur de l'ensemble. C'est pour cela que les grandes plateformes captent autant de valeur : elles jouent dans une économie où **l'effet de réseau, la scalabilité et l'automatisation renforcent le capital au lieu de l'éroder.**

C'est l'une des raisons pour lesquelles les GAFAM ont pris une si grande place dans l'économie mondiale, en seulement quelques années.

Le besoin d'un réveil européen

C'est d'ailleurs ce que souligne Mario Draghi, dans son fameux rapport. Il souligne que les USA et l'Union Européenne ont eu une évolution quasi identique pour ce qui est de l'industrie. La différence faramineuse qui nous sépare est notamment due à l'explosion des usages numériques, et à la place gigantesque, voir le monopole prise par les GAFAM.

Elles opèrent dans une économie fondée sur les rendements croissants, captent des milliards d'euros grâce à des modèles exportables à très faible coût marginal, mais **échappent largement**

aux systèmes fiscaux des pays dans lesquels elles opèrent. Elles vendent partout, imposent leurs standards, mais la valeur générée ne revient pas dans les territoires qui les utilisent. Nous les utilisons, nous les payons, nous leur donnons nos données, quant à elles, elles ramènent les profits au pays, manœuvrent pour éviter les impôts et taxes. **Et notre gouffre financier se creuse : l'argent sort et ne revient pas...**

La France et l'Europe ont pris beaucoup de retard sur ces sujets, restant mentalement et économiquement arrimés à son ancien monde industriel. A force de lobbying, nos décideurs ont fini par être persuadés que nous, Européens, n'étions pas capables de répondre à leurs besoins, que nous n'avions pas d'alternatives locales. Résultat : nous utilisons des logiciels venus d'ailleurs, en situation de monopole de leur marché, déployés sur des infrastructures, que nous ne maîtrisons pas. Le tout, venant de pays dont nous ne sommes plus sûrs de pouvoir appeler "alliés"

Passer de la prestation à la création de produits

Un autre levier important est celui de la création d'assets. Une grande partie du marché informatique français repose aujourd'hui sur des entreprises de services numériques (ESN). Ce modèle a ses avantages : il crée de l'emploi, il forme des profils variés, il rend des projets possibles. Mais il reste dans **une logique de flux** : on vend des jours-homme, on facture des missions, on répond à des appels d'offres.

Ce système, qui repose souvent sur l'optimisation de la ressource humaine, ne génère pas de propriété intellectuelle et est soumis au rendement décroissant du capital. Il crée de la dépendance économique à la commande et incite à la création de monopole pour simplifier les formations pour les équipes. Les éditeurs de logiciels, eux, travaillent dans une **logique d'assets**. Ils investissent dans des produits qui ont vocation à être réutilisés, distribués, perfectionnés. Leur modèle repose sur l'échelle, la capitalisation, l'automatisation. Ils ne vendent pas du temps, ils construisent du patrimoine technologique.

Aujourd'hui, la France a besoin des deux, mais **elle doit urgemment soutenir une économie basée sur la création de valeurs et des briques technologiques réutilisables** qui pourraient créer les leaders mondiaux de demain.

Le développeur, acteur stratégique de l'économie

Aujourd'hui, un ingénieur n'est pas un simple technicien. Il est au cœur de la transformation numérique et du bon fonctionnement de nos institutions, de nos entreprises, de nos services publics. **C'est lui qui construit les systèmes d'information, les plateformes logicielles, les chaînes de traitement de données qui font tourner l'économie moderne**.

Ce rôle est stratégique. Et pourtant, trop souvent, il reste cantonné à l'exécution, sans que l'on valorise sa capacité à structurer des visions technologiques à long terme. Dans mon métier, je vois chaque jour des développeurs capables de créer de nouveaux produits, de concevoir des architectures robustes, de prendre des décisions sur des choix d'outillage, de faire évoluer des systèmes complexes, de

projeter une vision sur le futur technologique du monde.

Ces compétences sont précieuses. Elles doivent s'exprimer dans des produits, pas seulement dans des projets. Il faut que davantage de techs puissent s'investir dans des aventures entrepreneuriales, dans des postes de décisionnaires, dans des logiciels pensés pour durer, pour être diffusés, pour créer de la valeur ici.

Soutenir les solutions locales, c'est construire l'écosystème

Aucune industrie ne prospère sans marché intérieur. Le numérique n'échappe pas à cette règle. Si nous voulons faire émerger des alternatives locales crédibles, il faut d'abord qu'elles soient utilisées, testées, challengées. Je ne vous dis pas qu'il faut tout le temps acheter local, de préférer systématiquement un produit français ou européen à un produit étranger : **il s'agit de lui donner sa chance**, en l'intégrant à un projet pilote, en faisant un retour d'expérience..

Aujourd'hui, trop de solutions françaises échouent à se développer faute de retours utilisateurs, de visibilité, ou d'adoption précoce. Or sans ces premières itérations, sans cet échange entre créateurs et utilisateurs, **aucun produit ne peut progresser** car c'est par l'usage que l'on construit des solutions solides. **Ne pas acheter tout de suite, c'est normal. Mais ne pas tester du tout, c'est se couper d'une partie de notre souveraineté**. En tant que développeur, vous avez ce pouvoir d'impulsion. En tant que décideur, le pouvoir de soutenir l'innovation.

Une économie forte passe par une industrie numérique forte

Le numérique est une industrie à part entière. Ce n'est pas une couche qu'on ajoute sur l'économie classique. C'est l'infrastructure invisible de tout ce que nous faisons. Il structure les chaînes logistiques, les interactions commerciales, la production de connaissance, les services publics.

Si nous voulons conserver notre niveau de vie, financer nos services publics, et maintenir un modèle économique équilibré, il faut maîtriser cette infrastructure.

Et pour cela, nous avons besoin d'ingénieurs, de développeurs, d'administrateurs systèmes, de concepteurs produits qui travaillent pour l'écosystème local.

Dans un monde où les États-Unis, l'Inde, la Chine ou Israël ont massivement investi dans leurs champions numériques, **la France ne peut pas se contenter de suivre**. Nous avons des talents. Nous avons des écoles. Nous avons des infrastructures. **Ce qui manque, c'est l'alignement entre les usages, la commande publique, les financements et les ambitions technologiques.** Et c'est aussi une question culturelle : valoriser la prise de risque, la création de logiciels, la construction d'actifs numériques.

Financer notre modèle de société passe par l'économie numérique

Si vous avez envie d'une école pour vos enfants, d'un hôpital pour vos parents, et des routes praticables, alors il faut qu'on soit capable de financer notre modèle de vie. Et pour financer ce modèle de vie, il faut produire de la valeur économique.

Pas seulement en exportant des biens physiques, mais en construisant les fondations numériques de demain. Ce sont les développeurs qui ont ce pouvoir. Ce sont les ingénieurs, les architectes systèmes, les créateurs de logiciels. Vous êtes au bon endroit, au bon moment. **L'économie de demain, c'est l'économie numérique. Et elle dépend de vous.**

De contributeur à mainteneur principal d'un driver de base de données Rust

Pierre Zemb
Staff Engineer @ Clever Cloud

clever cloud

Quand un composant logiciel critique devient soudainement orphelin, que faire ? C'est le défi auquel j'ai été confronté avec le driver Rust pour FoundationDB (foundationdb-rs), une librairie utilisée par mon employeur, Clever Cloud, et d'autres acteurs.

Cet article retrace une aventure : de la découverte de l'état du projet à la décision de créer un *fork*, en passant par les défis techniques de la maintenance d'un *binding* Rust autour d'une API C (la fameuse FFI), jusqu'à l'implémentation de stratégies de tests avancées, notamment l'intégration avec le puissant framework de simulation déterministe de FoundationDB. C'est un retour d'expérience concret sur la maintenance open source, la gestion des risques liés aux dépendances et l'importance cruciale des tests pour fiabiliser les systèmes distribués.

FoundationDB

L'état initial : un projet communautaire en déshérence

J'ai découvert rapidement que ce *binding* Rust n'était pas maintenu par l'équipe principale de FoundationDB, mais par la communauté. *"On voyait que c'était un peu bizarre, le repo GitHub se baladait d'organisation en organisation,"* ai-je noté. Pire encore, des fonctionnalités clés manquaient, et la toute première que je souhaitais utiliser était absente. Profitant du temps disponible, je me suis lancé : *"Bah allons-y, let's go, j'ai du temps à perdre, je vais aller contribuer."*

Le silence radio et le repo fantôme

J'ai soumis plusieurs *Pull Requests* (PRs) entre fin 2020 et mi-2021 pour ajouter les fonctionnalités manquantes et corriger

des problèmes. Cependant, mes contributions sont restées sans réponse. *"Pas de réponse… Tiens, c'est bizarre, j'ai pas de réponse, mais vraiment le calme plat."* L'activité sur le dépôt GitHub diminuait, la CI (Intégration Continue) a fini par tomber en panne, empêchant toute fusion de code. Le constat était sans appel : *"Le repo était complètement inactif. […] C'était limite, un repo mort."*
Pourtant, l'outil n'était pas mort pour tout le monde. Les statistiques de crates.io, le registre de paquets Rust, montraient des téléchargements quotidiens. Des développeurs, des entreprises même, utilisaient cette librairie devenue fantôme, maintenue par personne. Avec quelques autres contributeurs potentiels, nous nous sommes retrouvés face à un mur.
Une *issue* ouverte sur GitHub pour demander des nouvelles est restée, elle aussi, lettre morte. *"À ce moment-là, j'avoue, j'avais un peu abandonné,"* ai-je confié. Le déblocage est survenu de manière inattendue, par un email de l'ancien mainteneur.

Le contact inespéré et la décision du fork

J'ai appris que suite à son départ de l'entreprise hébergeant le dépôt GitHub, personne n'avait plus les droits nécessaires pour maintenir le code source à cet endroit. La situation semblait bloquée. Heureusement, deux éléments permettaient d'entrevoir une solution :

- L'ancien mainteneur conservait les droits de publication sur crates.io, essentiel pour diffuser de nouvelles versions.
- La licence open source (MIT/Apache 2.0) autorisait légalement la création d'un *fork* du projet.
- Face à cette impasse commune à plusieurs contributeurs, et devant le besoin croissant d'utiliser et de faire évoluer le driver pour nos projets chez Clever Cloud, la décision de réaliser un *hard fork* s'est imposée. Pour assurer une gouvernance plus stable et éviter les problèmes liés à l'hébergement par une seule entreprise, nous avons créé une organisation GitHub dédiée : foundationdb-rs.

Renaissance et restructuration technique

Le *fork* n'a pas été une simple copie. Ce fut l'occasion d'une remise à plat technique et organisationnelle :

- Migration du code vers la nouvelle organisation.
- Renommage de la branche principale (master vers main).
- Nettoyage et amélioration de la CI, devenue indispensable.
- Introduction de *tiers de compatibilité* pour clarifier le support des différentes plateformes (Linux x86_64 en Tier 1, macOS en Tier 2, etc.), abandonnant au passage le support de Windows, non maintenu par FoundationDB elle-même.
- Fusion des PRs en attente et ajout de nouvelles fonctionnalités.

Cette refondation a culminé avec la publication de la version 0.5.0 en avril 2022, marquant la renaissance officielle du projet sous une nouvelle gouvernance. L'ancien mainteneur m'a transféré les droits de publication sur crates.io, assurant la continuité.

Les défis techniques : tirer parti de Rust et maîtriser la FFI

Le choix de Rust pour développer un driver de base de données comme celui-ci n'est pas anodin. Ses garanties fortes en matière de sécurité mémoire (*memory safety*) et de gestion de la concurrence (*concurrency*), sans nécessiter de garbage collector, en font un candidat idéal pour construire des logiciels systèmes performants et fiables. L'objectif est d'offrir aux utilisateurs une API idiomatique Rust, sécurisée et agréable à utiliser.
Cependant, le driver Rust ne réimplémente pas toute la logique client de FoundationDB. Il s'appuie sur la bibliothèque C officielle, libfdb, fournie par les développeurs de FoundationDB. Pour faire communiquer le code Rust avec cette bibliothèque C, nous devons utiliser une interface appelée FFI (Foreign Function Interface).
La complexité inhérente à la FFI : C'est là que réside le principal défi technique. L'interaction entre Rust et C via la FFI nécessite l'utilisation de blocs unsafe. À l'intérieur de ces blocs, le compilateur Rust ne peut plus garantir automatiquement la sécurité mémoire comme il le fait pour le code Rust "pur". En tant que mainteneur, je dois donc vérifier manuellement que toutes les règles et préconditions d'utilisation de l'API C sont scrupuleusement respectées : gestion correcte des pointeurs, cycle de vie des objets C alloués, libération de la mémoire, respect des modèles de threads, etc. Avec environ 130 blocs unsafe disséminés dans le code du driver, le risque d'introduire des bugs subtils (fuites mémoire, *data races*, *segfaults*) est bien réel si l'on manque de rigueur. *"Le code Rust n'est pas une crate de base de données complète, il communique avec une bibliothèque C qui gère tout,"* rappelais-je. Cette dépendance à unsafe et à la correction du code C sous-jacent impose une vigilance constante.
L'arsenal de tests pour contrer la complexité : Pour garantir la fiabilité du driver malgré l'utilisation intensive de la FFI, une stratégie de tests exhaustive et multi-niveaux est absolument indispensable. C'est notre principal rempart contre les erreurs potentielles introduites par les blocs unsafe. Nous avons mis en place :

1. **Tests classiques étendus** : Nous exécutons une large suite de tests unitaires et d'intégration. Ces tests sont systématiquement lancés dans une grande variété d'environnements : sur plusieurs systèmes d'exploitation (Ubuntu, macOS), avec différentes versions de FoundationDB (de la 6.1 à la 7.3) et sur plusieurs versions du compilateur Rust (la version minimale supportée, stable, bêta et *nightly*). Tester sur la version *nightly* s'avère particulièrement utile pour détecter en amont d'éventuels changements de comportement du compilateur.
2. **Le BindingTester de FoundationDB** : L'outil le plus puissant à notre disposition est le BindingTester, développé par les mainteneurs de FoundationDB eux-mêmes. Il s'agit d'une suite de validation inter-langages conçue pour s'assurer que tous les *bindings* (Rust, Python, Go, etc.) se comportent de manière identique et correcte. Le BindingTester utilise une machine à états pour générer des séquences d'opérations aléatoires mais déterministes (grâce à une graine, ou *seed*). Ces opérations sont exécutées à la fois par notre *crate* Rust et par une implémentation de référence (en Python). Les résultats sont ensuite comparés pour détecter toute divergence. L'aspect déterministe permet de reproduire localement un scénario de test qui aurait échoué. Nous exécutons ce BindingTester en continu sur notre intégration continue (GitHub Actions), où chaque exécution déclenche l'exécution de milliers de scénarios de tests générés aléatoirement, assurant une validation constante de la conformité du *binding*.

Cette combinaison de FFI maîtrisée et de tests rigoureux est essentielle pour fournir une *crate* fiable aux utilisateurs.

Simulation déterministe : L'arme secrète de FoundationDB au service du code utilisateur

Les tests classiques et le BindingTester sont essentiels pour valider la conformité de notre *binding* Rust lui-même. Mais qu'en est-il de l'application qui *utilise* ce *binding* ? Comment s'assurer qu'elle se comportera correctement face aux aléas inhérents aux systèmes distribués en production : pannes réseau, machines lentes, bugs subtils qui n'apparaissent que sous une charge spécifique ou après une séquence d'événements particulière ?

C'est là qu'intervient l'approche qui, à mon sens, différencie radicalement FoundationDB et a profondément influencé ma vision de l'ingénierie logicielle : **la simulation déterministe**.

Qu'est-ce que la Simulation déterministe ?

FoundationDB est réputé non seulement pour sa base de données mais aussi, et peut-être surtout, pour son framework de simulation déterministe, développé avant même la base de données elle-même. Comme expliqué sur son site, l'objectif est de diagnostiquer les problèmes en simulation plutôt qu'en production.

> *"La simulation est capable de conduire une simulation déterministe d'un cluster FoundationDB entier au sein d'un processus mono-threadé. Le déterminisme est crucial car il permet une répétabilité parfaite d'une exécution simulée [...]. La simulation avance dans le temps [...] représentant une plus grande quantité de temps réel dans une quantité moindre de temps simulé."*

Concrètement, ce framework exécute une version complète du système distribué (la base de données, les clients) dans un seul processus, contrôlant l'ordonnancement des événements et le passage du temps. Il peut ainsi simuler et injecter de manière contrôlée et **reproductible** (grâce à une *seed* aléatoire) une immense variété de pannes : partitions réseau, latences variables, pannes de disque, machines qui tombent et reviennent, variations de topologie, corruption de données, etc. Le tout à une cadence bien plus élevée que dans le monde réel. C'est cet outil qui a permis à FoundationDB d'atteindre son niveau de robustesse exceptionnel.

Le fossé Développement vs Production

Cette approche s'attaque directement au fossé souvent observé entre l'environnement de développement contrôlé et la réalité chaotique de la production. En développement, nous testons des chemins "heureux" ou quelques erreurs simples. En production, les systèmes subissent des combinaisons complexes de pannes, des comportements utilisateurs imprévus, et des "cas limites" qui deviennent la norme à grande échelle (comme des pannes machines fréquentes sur de grands clusters). Les tests traditionnels peinent à couvrir cet espace exponentiel de possibilités. La simulation déterministe permet de l'explorer de manière systématique et automatisée, en testant le système face aux "pires" versions des utilisateurs et du monde.

Intégrer le code utilisateur dans la Simulation

La véritable innovation que nous avons exploitée avec foundationdb-rs est d'avoir réussi à intégrer du code applicatif écrit en Rust **directement à l'intérieur** de ce puissant framework de simulation C++. Grâce à une interface spécifique que nous avons développée, le trait RustWorkload, les développeurs peuvent désormais écrire leurs propres scénarios de test complexes en Rust (simulant la logique métier de leur application) et les exécuter *dans* l'environnement simulé de FoundationDB.

Cela signifie que le code applicatif Rust est soumis aux mêmes injections de fautes (réseau, disque, temps...), aux mêmes conditions de stress et bénéficie du même déterminisme que le cœur de FoundationDB lui-même lors de ses propres tests. Nous pouvons vérifier comment notre logique applicative réagit à une partition réseau pendant une transaction critique, ou à la perte soudaine de plusieurs nœuds de stockage, des scénarios difficiles voire impossibles à reproduire avec des tests d'intégration classiques.

*"Cela a été un **point clé majeur** pour nous permettre de développer et d'opérer Materia, l'offre de base de données serverless de Clever Cloud,"* expliquais-je. *"Nous pouvons profiter du même framework de simulation utilisé par les ingénieurs principaux de FDB pour notre propre code applicatif."* C'est une forme de *simulation-driven development* : utiliser la simulation non seulement pour valider a posteriori, mais aussi pour guider la conception même de systèmes résilients en amont.

Cette capacité à tester la logique applicative complète, et pas seulement le driver, dans des conditions de production simulées, contrôlées et déterministes, offre un niveau de confiance sans commune mesure avec les approches de test plus traditionnelles. C'est, à mon avis, l'un des apports les plus significatifs et différenciants de l'écosystème FoundationDB pour qui veut construire des systèmes distribués réellement fiables.

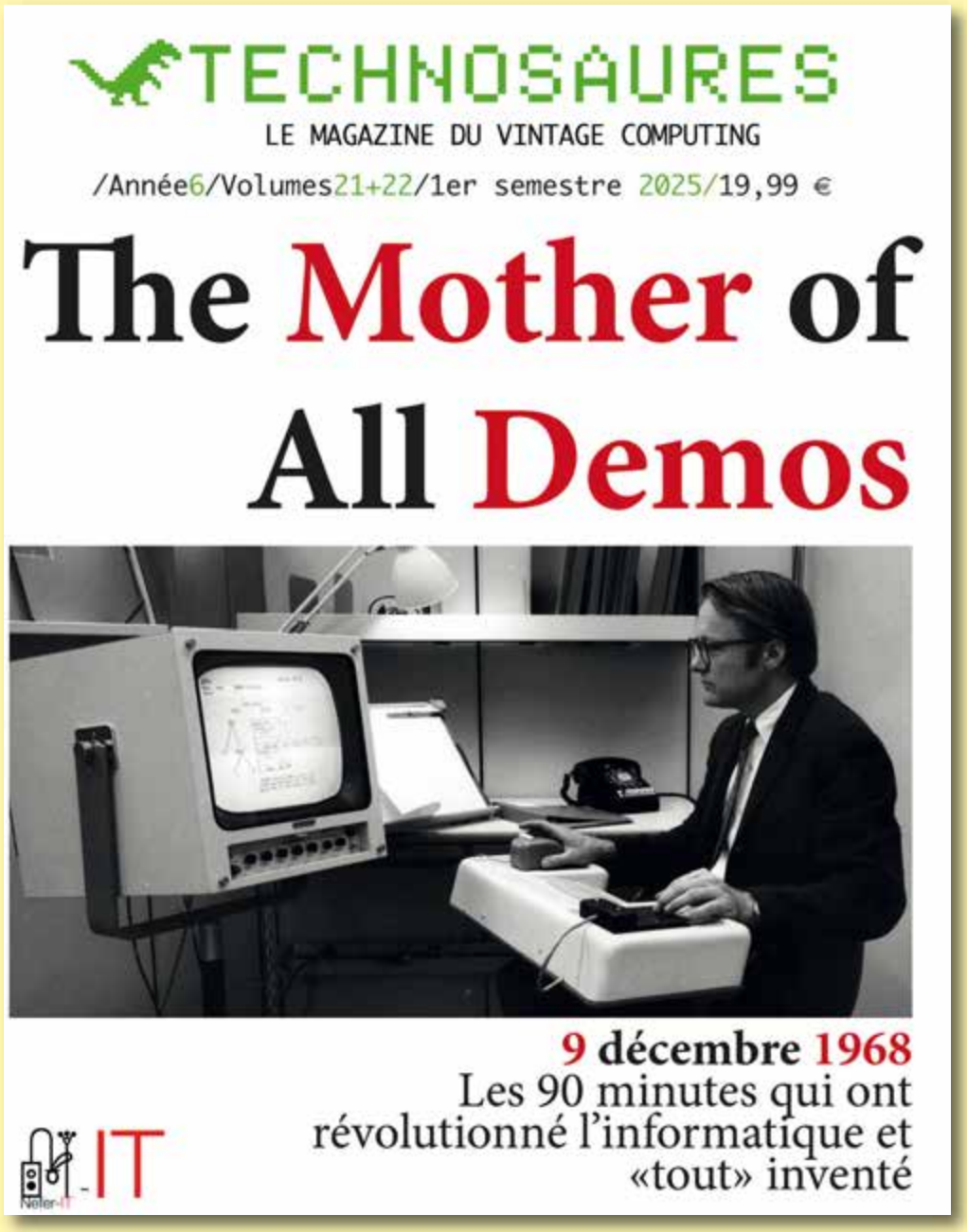
Conclusion et perspectives

Aujourd'hui, foundationdb-rs n'est plus un projet fantôme, mais un composant logiciel fiable, activement maintenu et essentiel pour plusieurs infrastructures critiques, y compris chez Clever Cloud. Cette renaissance illustre la capacité de l'écosystème open source à surmonter des défis majeurs, grâce à une combinaison d'engagement individuel, de soutien d'entreprise et, de manière cruciale, par l'adoption de méthodologies de test rigoureuses.

L'investissement conséquent dans la qualité, notamment via le BindingTester et l'intégration poussée avec le framework de simulation déterministe de FoundationDB, offre une confiance solide dans la robustesse du driver face aux complexités de la FFI et du monde distribué. Ces outils, initialement conçus par et pour les équipes de FoundationDB, se sont révélés essentiels : ils rendent la maintenance de ce code complexe gérable par une seule personne en automatisant une grande partie de la validation. C'est cette approche qui permet de sécuriser les usages actuels et d'envisager sereinement l'avenir.

Mon expérience avec ce projet renforce ma conviction : les techniques de tests avancées comme la simulation déterministe sont des outils extraordinairement puissants, bien que parfois sous-estimés, pour construire et maintenir les systèmes distribués fiables dont dépendent nos services numériques. Bien que le travail de maintenance et d'amélioration soit continu, les fondations techniques et la gouvernance du projet sont désormais établies pour durer. L'aventure foundationdb-rs continue, avec la promesse d'explorer plus encore le potentiel de la simulation pour l'ingénierie logicielle de demain.

Le nouveau numéro de Technosaures est disponible !



Au sommaire :

- The Mother of All Demos
- Il y a 50 ans, l'Altair 8800
- GeOS tente de concurrencer Windows
- Falcon 030 ne sauve pas Atari

Disponible sur programmez.com et amazon.fr

Cas d'usage : développer une application IA de santé en toute souveraineté avec Cloud Temple

L'intelligence artificielle révolutionne le secteur de la santé : elle ouvre des perspectives inédites pour l'aide au diagnostic et l'amélioration de la relation patient-médecin. Mais, si sa montée en puissance ouvre la voie à des innovations de rupture, elle impose aussi une vigilance accrue sur la protection des données personnelles et leur souveraineté. Pour illustrer ce double défi, nous avons choisi d'endosser le rôle d'un développeur missionné pour créer une application IA médicale innovante tout en respectant le RGPD et les exigences françaises de sécurité. Ce cas d'usage présente ainsi le développement de "My Health Coach", une application d'assistance médicale intelligente, déployée sur l'écosystème cloud souverain de Cloud Temple.

Alexandru Lata

Alexandru façonne l'innovation technologique de Cloud Temple avec une conviction forte : l'IA, le cloud et la cybersécurité ne créent de valeur que s'ils servent des objectifs métiers clairs, dans un cadre de confiance. En tant que Chief Technology & Innovation Officer, il transforme ces technologies en produits concrets, performants et durables, au service des enjeux stratégiques des clients.

Le défi de l'IA en santé

Les données de santé sont parmi les plus sensibles qui existent et nécessitent un niveau de protection maximal. Le RGPD impose des contraintes strictes sur leur traitement, avec des sanctions pouvant atteindre 4% du chiffre d'affaires annuel. La certification HDS (Hébergement de Données de Santé) est par ailleurs obligatoire en France pour tout système manipulant ces informations, imposant des exigences techniques et organisationnelles drastiques.

Une problématique qui se complexifie encore avec les solutions cloud traditionnelles. En effet, les hyperscalers, malgré leur maturité technique indéniable, posent des questions fondamentales de souveraineté numérique. Le CLOUD Act américain permet par exemple aux autorités US d'accéder aux données hébergées par leurs entreprises, même sur le territoire européen, sans notification des autorités locales. À cela s'ajoute la loi FISA (Foreign Intelligence Surveillance Act), qui autorise les services de renseignement américains, via la section 702, à obtenir des données auprès des fournisseurs de cloud américains, y compris celles stockées en Europe, dans le cadre d'enquêtes de sécurité nationale – le tout sous le sceau du secret et sans que les personnes concernées ne soient informées. Une réalité juridique qui se révèle incompatible avec les exigences de confidentialité médicale françaises et européennes.

Face à ces contraintes, les développeurs se trouvent souvent dans une impasse technique et juridique : choisir entre innovation technologique et conformité réglementaire. Les solutions on-premise traditionnelles offrent le contrôle total, mais limitent drastiquement l'agilité de développement et l'accès aux technologies IA de pointe. Dans ce contexte l'utilisation d'un cloud souverain s'impose comme le meilleur des deux mondes : un cloud souverain alliant sécurité maximale et innovation technologique, sans compromis sur l'expérience développeur.

Le cas d'usage

Pour démontrer concrètement les capacités de l'écosystème Cloud Temple, prenons l'exemple théorique de "My Health Coach", une application d'assistance médicale intelligente qui pourrait révolutionner la préparation des rendez-vous médicaux. Cette application fictive, créée pour démontrer nos propos, analyserait les dossiers médicaux des patients grâce à l'IA et générerait automatiquement des questions pertinentes et personnalisées à poser lors des consultations médicales.

Le processus utilisateur est volontairement simple et intuitif : le patient télécharge ses documents médicaux (analyses biologiques, comptes-rendus d'imagerie, ordonnances, lettres de spécialistes) de manière totalement sécurisée via une interface web responsive. L'IA analyse ces informations en tenant compte de l'historique médical complet et du contexte spécifique du prochain rendez-vous médical. Elle génère ensuite une liste de questions personnalisées et contextualisées : "Docteur, mes dernières analyses montrent une légère hausse du cholestérol LDL, faut-il adapter mon traitement actuel ?" ou "J'ai ressenti des vertiges avec mon nouveau médicament, existe-t-il des alternatives thérapeutiques ?".

L'interface conversationnelle avancée permet au patient d'affiner ces questions, d'en ajouter de nouvelles ou de demander des explications détaillées sur ses résultats médicaux. L'objectif est double : optimiser le temps précieux de consultation en préparant les bonnes questions et améliorer significativement la communication patient-médecin en donnant au patient les clés pour mieux comprendre sa santé et participer activement à son parcours de soins.

La réponse Cloud Temple

Cloud Temple propose un écosystème cloud souverain français qui résout cette équation complexe. Son cloud de confiance, qualifié SecNumCloud par l'ANSSI, offre le plus

haut niveau de sécurité disponible en France, tout en conservant l'agilité des clouds modernes.
Pour ce développement, nous avons choisi deux briques complémentaires de l'écosystème Cloud Temple :

- Le PaaS OpenShift, qui fournit un environnement Kubernetes managé avec tous les outils DevOps intégrés.
- Le service LLMaaS, qui donne accès à 36 modèles d'IA générative, incluant des modèles français comme Mistral, tous hébergés en France dans l'environnement cloud de confiance de Cloud Temple.

Cette approche présente un avantage décisif : les données ne quittent jamais le territoire français et restent soumises exclusivement à la loi française. La qualification SecNumCloud garantit que l'infrastructure respecte les standards de sécurité les plus élevés, validés par l'État français.

L'écosystème Cloud Temple

Le PaaS OpenShift de Cloud Temple transforme la complexité de Kubernetes en simplicité d'usage. Basé sur Red Hat OpenShift, il intègre nativement les outils essentiels : registre Quay sécurisé, pipelines CI/CD GitLab, GitOps avec ArgoCD, monitoring Prometheus/Grafana, logging ELK et service mesh Istio.

L'avantage majeur ? L'absence de vendor lock-in : les applications restent portables vers tout environnement Kubernetes standard, garantissant la pérennité des investissements.

Le stockage S3 compatible HDS complète l'offre avec une solution objet sécurisée en zone SecNumCloud. Cette infrastructure répond aux exigences HDS avec chiffrement, redondance et traçabilité complète, tout en conservant l'API S3 standard pour faciliter l'intégration.

Le service LLMaaS propose un catalogue de 36 modèles d'IA générative via API REST standardisée : Qwen3 30B pour le français médical, DeepSeek-R1 671B pour les analyses complexes, Qwen3 235B pour les synthèses, ou Llama pour plus de contrôle.

Cette approche API-first simplifie l'intégration IA : plus besoin de gérer l'infrastructure GPU, les mises à jour de modèles ou l'optimisation. L'API Cloud Temple gère automatiquement montée en charge, répartition entre modèles et quotas utilisateurs.

L'architecture de l'app

L'architecture suit les bonnes pratiques cloud-native modernes. Le frontend React/TypeScript offre une interface responsive. Il communique avec un backend Node.js via une API REST sécurisée, qui orchestre les interactions avec PostgreSQL et les services d'IA.

Cette architecture en micro-services facilite la maintenance, car chaque composant peut être développé et déployé indépendamment. PostgreSQL stocke les profils patients et l'historique médical, le tout chiffré avec AES-256.

L'intégration avec LLMaaS s'effectue via des appels API REST sécurisés. Le backend prépare les prompts médicaux, appelle le modèle IA approprié, puis traite la réponse. Cette approche permet de maintenir un contrôle total sur les données.

Enfin, la conteneurisation Docker garantit la cohérence entre environnements. Le déploiement sur Kubernetes via OpenShift automatise la gestion des ressources, le scaling et la haute disponibilité.

Assurer une sécurité multi-niveaux

La sécurité repose sur une approche de défense en profondeur. Le chiffrement TLS 1.3 sécurise toutes les communications. Au niveau stockage, les données sensibles sont chiffrées avec AES-256. La gestion des clés suit les bonnes pratiques de sécurité en préconisant l'utilisation de services de secrets externes. L'injection sécurisée s'effectue via des solutions comme External Secret Operator, permettant une séparation claire entre les secrets et l'infrastructure Kubernetes tout en maintenant un contrôle d'accès RBAC granulaire.

La micro-segmentation réseau isole les composants et les Network Policies Kubernetes définissent précisément les communications autorisées. L'audit est assuré par un logging centralisé : connexions, accès aux données et appels API sont ainsi tracés.

La conformité RGPD est intégrée dès la conception avec l'implémentation du droit à l'oubli et la portabilité des données. La certification HDS de Cloud Temple garantit quant à elle le respect des exigences spécifiques aux données de santé.

Déployer sur le PaaS OpenShift de Cloud Temple

Le déploiement illustre la simplicité d'OpenShift :

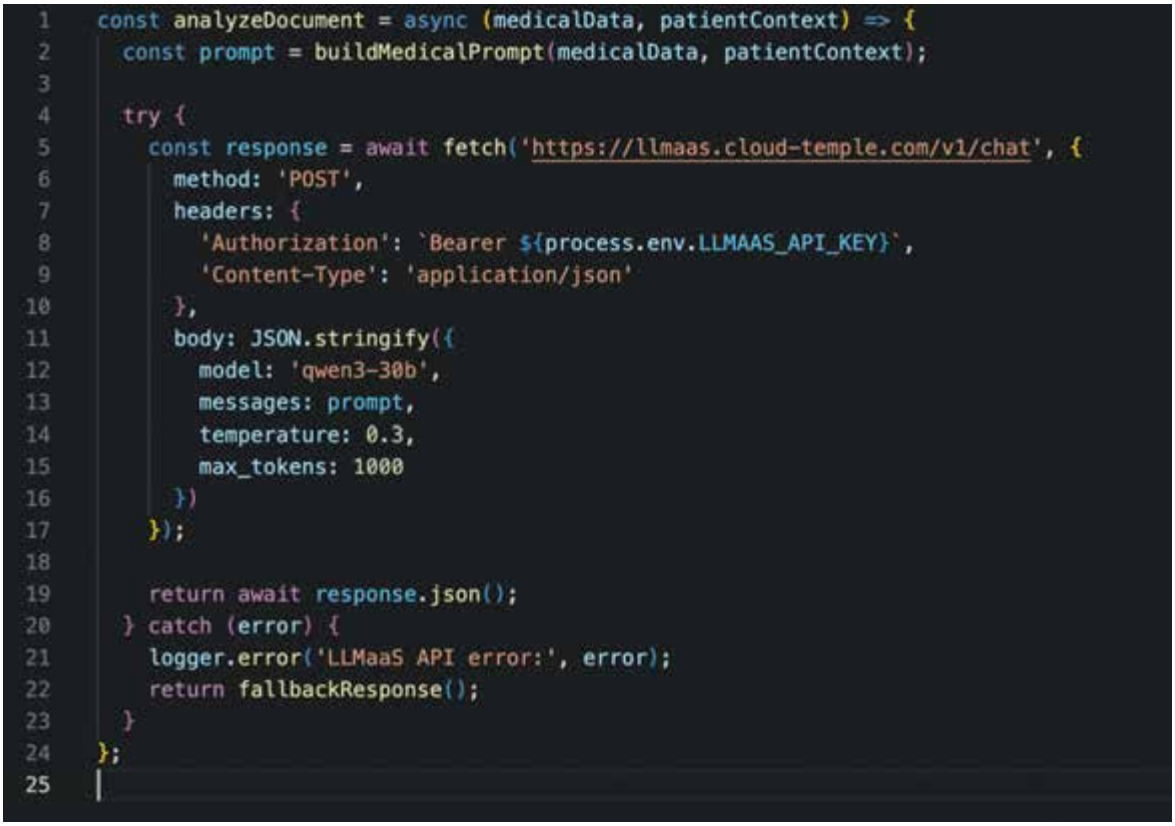
- Le pipeline CI/CD GitLab automatise le processus : à chaque commit, le code est testé, les images Docker construites, scannées pour les vulnérabilités, puis déployées.
- Les Helm Charts standardisent la configuration Kubernetes.
- Le monitoring Prometheus collecte les métriques.
- Grafana visualise les données avec des dashboards préconfigurés.
- L'auto-scaling ajuste automatiquement le nombre de pods selon la charge.

Cette automatisation libère l'équipe des tâches d'infrastructure pour se concentrer sur la valeur métier de l'application. Le déploiement se résume à un git push.

Intégrer l'IA générative avec le LLMaaS de Cloud Temple

Pour notre application, le choix s'est porté sur Qwen3 30B, un modèle performant optimisé pour les textes médicaux et capable de traiter efficacement le français.

L'API REST suit les standards OpenAI, ce qui facilite l'intégration. L'authentification par Bearer Token sécurise les accès. Le prompt engineering médical intègre le contexte médical et les contraintes déontologiques françaises.
L'exemple de code suivant illustre l'intégration : **Figure A**



```
const analyzeDocument = async (medicalData, patientContext) => {
  const prompt = buildMedicalPrompt(medicalData, patientContext);

  try {
    const response = await fetch('https://llmaas.cloud-temple.com/v1/chat', {
      method: 'POST',
      headers: {
        'Authorization': `Bearer ${process.env.LLMAAS_API_KEY}`,
        'Content-Type': 'application/json'
      },
      body: JSON.stringify({
        model: 'qwen3-30b',
        messages: prompt,
        temperature: 0.3,
        max_tokens: 1000
      })
    });

    return await response.json();
  } catch (error) {
    logger.error('LLMaaS API error:', error);
    return fallbackResponse();
  }
};
```

Figure A

Retour d'expérience

Cet exemple théorique démontre la puissance de l'écosystème Cloud Temple pour développer rapidement des applications IA dans des environnements hautement sécurisés. Bien que cette application n'existe pas en production, elle montre comment le développement d'une telle solution, intégrant IA et conformité médicale, est facilité par les briques OpenShift et LLMaaS de Cloud Temple et peut transformer l'expérience des développeurs.

Premier avantage notable : la rapidité de développement. Grâce au PaaS OpenShift, le passage de l'idée au prototype fonctionnel s'effectue en quelques jours seulement. Les outils DevSecOps intégrés permettent de mettre en place automatiquement les pipelines CI/CD, le monitoring et la sécurité, sans configuration manuelle complexe. Un développeur peut ainsi se concentrer exclusivement sur la logique métier de son application.

L'intégration de l'IA via LLMaaS vient révolutionner l'approche traditionnelle. Là où l'implémentation d'un modèle IA nécessitait auparavant des semaines de configuration d'infrastructure GPU et d'optimisation, quelques lignes de code suffisent désormais. L'API standardisée permet d'expérimenter avec différents modèles (Qwen3, DeepSeek) sans refactoring majeur, accélérant considérablement les phases de prototypage et d'optimisation.

La sécurité by design élimine les frictions habituelles entre innovation et conformité. Contrairement à nombre d'idées reçues sur les environnements hautement sécurisés, l'environnement SecNumCloud n'impose aucune contrainte supplémentaire au développeur. Les outils restent identiques à un environnement cloud classique, mais l'infrastructure sous-jacente garantit automatiquement la conformité RGPD, HDS et SecNumCloud.

L'approche DevSecOps native de Cloud Temple transforme également la gestion des déploiements. Chaque commit déclenche automatiquement les tests de sécurité, la construction d'images Docker durcies et le déploiement sécurisé. Cette automatisation élimine les erreurs humaines et garantit un niveau de sécurité constant, même lors de déploiements fréquents.

Enfin, les performances obtenues valident l'approche : latence inférieure à 200ms, disponibilité de 99.9%, scaling automatique selon la charge. Ces métriques, équivalentes aux meilleures plateformes mondiales, prouvent que souveraineté numérique et performance technique ne sont plus antinomiques. La différenciation concurrentielle devient tangible pour les développeurs travaillant sur des applications sensibles.

Pouvoir garantir que les données restent en France, sous juridiction française, avec un niveau de sécurité validé par l'État : un avantage qui se révèle décisif face aux solutions extra-européennes, particulièrement dans les secteurs régulés comme la santé, la finance ou la défense.

Ce cas d'usage démontre qu'il est possible de concilier souveraineté numérique et innovation technologique. L'écosystème PaaS OpenShift + LLMaaS offre aux développeurs français une alternative crédible aux hyperscalers, sans compromis sur les fonctionnalités.

Raphaël Nicoud
Président d'Aqua Ray
Raphaël créé en 2003 avec Guillaume de Lafond : Aqua Ray. L'entreprise propose des services d'hébergement de serveurs informatiques (physiques ou virtuels), de centres de données, de conception d'infrastructures et d'infogérance.

Cloud souverain : comment l'open source peut garantir une indépendance technique ?

À l'heure où les enjeux de souveraineté numérique s'imposent dans tous les débats, les développeurs sont plus que jamais aux avant-postes de la transformation technologique. Entre dépendance aux géants de la Tech américaine et quête d'un modèle plus autonome, la question du cloud souverain n'est plus théorique : elle devient une nécessité. Et dans cette quête, l'open source s'avère être un levier essentiel. On vous explique pourquoi !

VMware, Unity, Google, Oracle : des exemples qui doivent alerter

L'histoire récente du numérique regorge d'exemples où la dépendance à des solutions fermées s'est transformée en piège. On peut citer le rachat de VMware par Broadcom qui a été un électrochoc sur le marché de la virtualisation. Explosion des coûts de licence, restructuration de l'offre, incertitudes techniques. Du jour au lendemain, des équipes DevOps ont dû envisager des migrations en urgence, faute de pouvoir absorber les nouvelles conditions commerciales.

Même scénario pour Unity, qui a modifié à deux reprises les conditions d'utilisation de son moteur en 2023 et 2024, impactant brutalement les développeurs. De nombreux projets ont dû migrer vers Godot pour retrouver de la maitrise.

Vous pensez être à l'abri de ces scénarios ? Rappelons qu'Oracle a modifié en 2019 les conditions de licence de Java SE, forçant de nombreux systèmes Java en production à migrer vers des alternatives open source sous contrainte de temps. Et que Google a annoncé en 2025 une nouvelle segmentation de ses APIs Maps, réduisant les quotas gratuits sur des services utilisés "par défaut" dans nombreux projets Web.

L'open source : un levier de maîtrise

Loin des discours idéalistes, l'open source est aujourd'hui un choix rationnel. Il garantit non seulement une auditabilité du code, mais il offre également la liberté de comprendre, de modifier, et de faire évoluer ses outils sans attendre la prochaine décision d'un comité exécutif à Palo Alto ou Redmond. Et dans un contexte où les exigences de sécurité se renforcent — NIS2, ISO 27001, et SecNumCloud pour la France — cette auditabilité devient un critère de plus en plus exigé.

Des solutions concrètes montrent à quel point le libre permet de reprendre la main sur des composants critiques. Prenons l'exemple de Keycloak, utilisé comme solution d'authentification open source, il permet de remplacer des services comme Auth0 tout en gardant un contrôle total sur le comportement de l'outil (intégration SSO, personnalisation, hébergement interne).

Construire une infrastructure libre et cohérente

Chez Aqua Ray, nous avons fait le choix radical du 100 % libre : des hyperviseurs comme XCP-ng, nés né d'une volonté d'indépendance vis-à-vis de Citrix ou d'autres éditeurs comme VMware, aux systèmes de stockage distribués comme CEPH, jusqu'aux outils d'exploitation et de monitoring, tous maîtrisés en interne. Il existe d'ailleurs d'autres choix très valables, comme Proxmox, qui démontrent qu'il est possible de bâtir des infrastructures robustes sans chaînes invisibles. Cette cohérence nous garantit qu'aucune décision technique ne soit dictée par un fournisseur tiers. Nous sommes convaincus que l'open source est une assurance-vie technologique.

La boîte à outils du développeur moderne

En tant que développeurs, vous êtes en première ligne. C'est à vous que revient la responsabilité de choisir les bons outils, de concevoir des architectures résilientes, et de penser long terme. Voici les 4 principales raisons de passer au libre :

- L'auditabilité : elle n'est pas seulement un confort intellectuel mais une exigence stratégique. Dans un contexte de cybersécurité accrue, pouvoir vérifier chaque ligne de code, chaque protocole de communication, chaque module utilisé dans une infrastructure, est un atout majeur.
- La personnalisation : un développeur doit pouvoir adapter ses outils à ses besoins, sans dépendre d'un éditeur. L'open source le permet : chaque brique peut être modifiée, optimisée, et réadaptée à une nouvelle architecture.
- Le coût : il joue évidemment un rôle essentiel. Non pas uniquement par la gratuité apparente mais aussi par la pérennité et la maîtrise des évolutions. Un projet bâti sur des standards ouverts et des outils libres sera plus robuste et plus facile à maintenir. De plus, l'open source permet un alignement des coûts sur l'usage réel, évite les surcoûts inutiles et protège contre les hausses imposées par des logiques purement financières.
- Enfin, la pérennité : l'histoire regorge d'exemples de solutions propriétaires abandonnées du jour au lendemain. Adobe Flash, Microsoft Silverlight, Google Cloud IoT Core, toutes fermées malgré leur succès initial. Miser sur l'open source et choisir un hébergeur aligné sur ces principes, qui offre une infrastructure libre et souveraine, est une garantie contre ces changements arbitraires.

En conclusion

Vous l'aurez compris, le cloud souverain ne se décrète pas : il se construit. Et l'open source est aujourd'hui le socle le plus crédible pour cette construction. Il offre aux développeurs un cadre pour innover librement, tout en contribuant à réduire le lock-in et en garantissant une vraie indépendance technique. Pour nous, le logiciel libre n'est pas un dogme, c'est un état d'esprit : celui d'une Europe numérique souveraine, éthique et durable.

En direct des actualités programmez.com

LA SÉMANTIQUE DES VERSIONS

Version 3.0, version 3.1.125, version bêta, RC, GM, etc. Il est toujours utile de revenir aux fondamentaux de la sémantique des versions. Prenons un exemple : version 1.1.20

1.x.x -> numérotation de la version majeure

x.1.x -> numérotation mineure

x.x.20 -> patch, bug fix

Dans cet exemple, nous avons 3 éléments. La version majeure change quand les changements sont importants. Dans ce cas, on justifie le changement de numéro. Dans le cas de Python, nous avons la release 2 et la release 3. Le second numéro concerne le plus souvent les mises à jour dites mineures. Ce sont souvent des ajustements, des fonctionnalités qui ne sont pas considérées comme majeures. Le dernier numéro est souvent utilisé pour les mises à jour de sécurité, des corrections de bugs, etc. Généralement, il n'y a pas de nouvelles fonctionnalités.

Il existe parfois d'autres sous-versions.

Au-delà, on parle d'alpha, preview, bêta, RC, release, GA, build. Cela indique un état non final de la version. Prenons un exemple : OpenJDK 25 build 19.

OpenJDK : nom du langage, du logiciel, etc.

25 : version majeure

build 19 : indique une pré-version en développement et c'est la 19e version buildée et distribuée.

Il est même possible d'ajouter à la numérotation un état du développement : preview, b pour bêta ou rc pour release candidate.

Pour certains projets, on parlera de canaux (channels) de versions. Chromium utilise sur les versions desktop 4 canaux différents. Chaque canal correspond à un état de développement :

- canary : pré-version buildée chaque nuit, réservée aux développeurs expérimentés
- dev : version plus avancée et plus stable
- bêta : version plus table et complète pour tests élargis, mais pas pour la production
- stable : branche de la version finale et stabilisée pour tous les utilisateurs

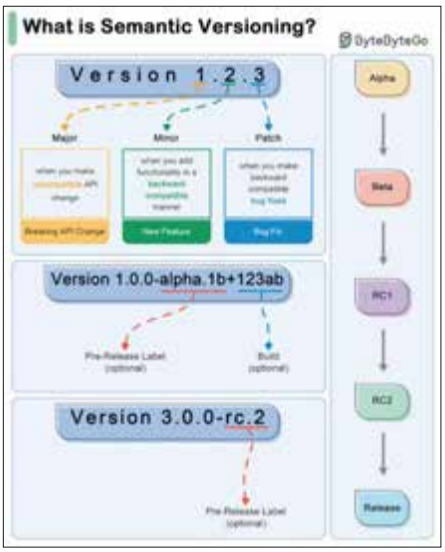
Revenons à nos alpha, bêta, RC, release / GA / GM. Les alpha, bêta ou preview sont les versions en développement pour une audience plus ou moins élargie. La RC (release candidate) est une version pré-finale : les fonctionnalités sont complètes et il s'agit de trouver et de fixer les derniers bugs. Plusieurs RC peuvent être distribuées. L'ultime étape est la version finale qui peut prendre différents noms selon le logiciel ou le projet : release, GA, GM, stable. Sur Azure, on parlera de GA (disponibilité générale) et non de release finale ou de version 1.0.

Attention : l'identifiant de version majeure zéro (0.y.z) est destiné au développement initial. Tout ou partie peut être modifié à tout moment. L'API publique ne devrait pas être considérée comme stable. (règle 4 du semver)

La définition du semver.org

Étant donné un numéro de version MAJEUR.MINEUR.CORRECTIF, il faut incrémenter :

1. le numéro de version MAJEUR quand il y a des changements non rétrocompatibles,

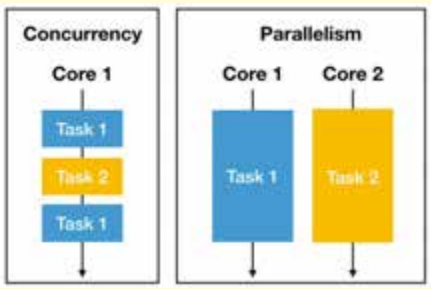


2. le numéro de version MINEUR quand il y a des ajouts de fonctionnalités rétrocompatibles,
3. le numéro de version de CORRECTIF quand il y a des corrections d'anomalies rétrocompatibles.

Des libellés supplémentaires peuvent être ajoutés pour les versions de pré-livraison et pour des métadonnées de construction sous forme d'extension du format MAJEURE.MINEURE.CORRECTIF.

Programmation concurrence / programmation parallèle ? Ben non

Dans la programmation moderne, nous parlons souvent de concurrence (concurrency) et de parallélisme. Nous avons parfois tendance à les confondre. Sauf qu'il s'agit de concepts différents avec un fonctionnement différent.



Concurrence / programmation concurrence / concurrency

La concurrence est une application capable de traiter plusieurs tâches en même temps. La concurrence réduit le temps de réponse du système en utilisant une seule unité de traitement. Nous avons une illusion de parallélisme, mais dans la réalité, non. Les tâches ne sont pas traitées en parallèle et une nouvelle tâche commence à être traitée même si la précédente n'est pas totalement terminée. La concurrence est un entrelacement des processus sur un même cœur / CPU. Pour résumer, la programmation concurrente est un paradigme de programmation tenant compte de plusieurs contextes d'exécution (threads, processus, tâches). La programmation concurrente est dite asynchrone, car une tâche est lancée avant que la précédente ne se termine. La concurrence permet de ne pas bloquer une application, de traiter les tâches demandées, et de pouvoir utiliser les processus multi-cœur. On comprend alors que la concurrence ne permet pas un traitement "en continu" de la tâche, car le système doit faire avancer chaque tâche.

Programmation parallèle

Le parallélisme est un modèle de programmation différent. Pour faire simple, il s'agit d'exécuter x tâches / processus sur autant de cœurs / de processeurs. Dans ce cas, le traitement est "linéaire" et chaque processus s'exécute sur un cœur, un processeur. Cette approche permet de mieux utiliser l'ensemble des processeurs et des cœurs. Une définition possible : Le parallélisme vise à accroître la vitesse de calcul grâce à l'utilisation de plusieurs processeurs. Il s'agit d'une technique permettant d'exécuter simultanément différentes tâches. Elle implique plusieurs unités de traitement indépendantes, ou dispositifs informatiques, fonctionnant et exécutant des tâches en parallèle afin d'accroître la vitesse de calcul et d'améliorer le débit. (d'après TechDifferences.com). Le parallélisme permet de lancer et de traiter des tâches variées en même temps sur x cœurs / CPU et chaque processus avance indépendamment de l'autre et sans ralentir les autres tâches. Dans cette approche, nous ne changeons pas de tâches. Et chaque cœur termine la tâche en cours. Il est possible de faire de la concurrence dans une approche parallèle.

Stéphane Carrez

CTO de Twinlife au sein du groupe Skyrock

Ingénieur diplômé de l'ISEP (1990), Stéphane est passionné par les systèmes embarqués et temps réel. Il a travaillé sur de nombreux projets industriels, notamment chez Sun Microsystems (ChorusOS) et Bouygues Telecom (Bbox). Il a aussi contribué par le passé au projet Open Source GCC, en ajoutant le support des microcontrôleurs 68HC11/12. Aujourd'hui, il est CTO de Twinlife, où il pilote le développement des messageries sécurisées twinme et Skred, conçues pour respecter la vie privée des utilisateurs. Stéphane a aussi lancé une vingtaine de projets Open Source en Ada, avec une attention particulière à la fiabilité et à la sécurité logicielle. Il est secrétaire de l'association Ada France.

Ada, le langage qui structure votre code et vos idées

Après de longues années d'absence, Ada vient de refaire son apparition dans le TIOBE Programming Community Index, l'index de popularité des langages de programmation. Pourquoi un tel regain d'intérêt, là où bien souvent les développeurs se tournent vers la facilité ? Une partie de la réponse se trouve dans la fiabilité que ce langage va ensuite apporter aux programmes. Découvrons les avantages qu'apporte Ada à travers un exemple didactique et ludique : celui du tic-tac-toe, jeu indémodable peut-être davantage connu sous le nom de morpion. Les sources sont disponibles en téléchargement sous plusieurs versions montrant différentes progressions dans l'implémentation. L'intégralité des sources n'est donc pas indiquée dans l'article afin de mettre l'accent sur les particularités d'Ada.

Un peu d'histoire

Ada trouve son origine suite à un appel d'offres lancé par le département de la Défense des États-Unis (DoD) dans les années 1975-1977 et remporté par l'équipe dirigée par Jean Ichbiah chez CII Honeywell Bull en 1979-1980. En 1983, la première norme ANSI/MIL-STD-1815A est déposée et c'est ainsi que naît Ada 83, en hommage à Ada Lovelace, considérée comme la première programmeuse de l'histoire.

Alors en avance sur les autres langages, Ada 83 intègre déjà des concepts de généricité et de multitâche. Presque tous les dix ans, Ada connaît des évolutions majeures : la programmation orientée objet et le temps réel avec Ada 95, le renforcement de l'orienté objet et la modernisation de la syntaxe en 2005. L'évolution suivante avec Ada 2012 marque un renforcement de l'axe de la sûreté avec l'ajout massif de contrats de programmation (préconditions, postconditions, invariants), inspirés du paradigme *Design by Contract*. Enfin, si la dernière version, Ada 2022, reste toujours très axée sur la sécurité et la fiabilité, elle propose quelques syntaxes plus modernes et simplifiées.

Mais quels avantages ?

Ada a été conçu pour écrire du code fiable, sûr et maintenable. Un développeur passe souvent plus de la moitié de son temps à comprendre, analyser ou corriger du code plutôt qu'à en écrire. Grâce à un langage lisible (même si on ne le connaît pas !), un compilateur qui réalise davantage de vérifications et permet de prouver formellement certaines parties, on réduit fortement les erreurs des développeurs.

Ada n'est en général pas destiné aux logiciels Kleenex, mais aux logiciels devant être maintenus sur de longues périodes. Citons le logiciel de 2,3 millions de lignes (voir annonce du FOSDEM 2020) créé par Eurocontrol qui depuis 20 ans permet de gérer le trafic aérien en Europe. Particulièrement utilisé dans des secteurs tels que l'aéronautique (Airbus), le spatial (Ariane), les radars (Thales) ou encore le ferroviaire, Ada est idéal pour mener à bien des projets conséquents et fortement confidentiels.

Étant donné qu'Ada utilise essentiellement des mots clefs en anglais pour définir les règles du langage, nous avons choisi pour notre exemple des identifiants anglais. Ada 2022 totalise ainsi 74 mots réservés. Peu de caractères spéciaux sont utilisés, facilitant la lisibilité du langage pour ceux qui ne connaissent pas Ada.

Une syntaxe évitant les confusions

La syntaxe Ada essaie d'éliminer les confusions afin de réduire les erreurs de programmation. Qui ne s'est pas trompé dans une comparaison en C sur le '==' en la remplaçant par une affectation '=' simplement en ayant oublié le second '=' ? Qui ne s'est pas perdu dans la fin d'une fonction en cherchant le dernier '}' marquant la fin du dernier bloc de celle-ci ?

Pour éviter cela, l'affectation est définie avec ':=' et ne peut pas être utilisée dans une condition. La fin des procédures, des fonctions et de certaines constructions est indiquée par le mot clef **end** suivi du nom de la procédure ou fonction : c'est l'équivalent du '}' dans plusieurs langages. Redondante et lourde en apparence, cette syntaxe force le développeur à être plus clair et précis dans ses intentions.

Démarrons avec Alire

Depuis quelques années, l'écosystème Ada s'est enrichi de nombreux outils, du compilateur (GCC GNAT) au debugger (GDB) en passant par les éditeurs (GNAT studio, Visual Studio). Désormais, un gestionnaire de paquets simplifie également la prise en main et la création de projets Ada : Alire

Devenu un outil indispensable pour démarrer rapidement et facilement un projet Ada, Alire permet de récupérer le compilateur, le debugger, l'assistant de preuve ainsi que toutes les bibliothèques et outils spécifiques proposés par la communauté Ada.

Vous pouvez récupérer le paquet d'installation d'Alire pour votre machine depuis https://ada-lang.io. Le site propose des installations pour Windows, GNU/Linux et macOS. Alire est aussi disponible pour FreeBSD, NetBSD et OpenBSD, mais

aucun binaire prêt à l'emploi n'est proposé. L'installation du paquet Alire va vous donner un exécutable unique alr disposant de plus de 20 commandes.
Pour démarrer notre projet, nous allons utiliser Alire et sa commande alr init. Dans notre cas, l'option —bin signifie que nous allons créer un exécutable et l'option —no-test que nous ne demandons pas de création de tests (pour simplifier). Nous indiquons ensuite à la commande le nom du projet.

```
alr init —bin —no-test tictactoe
```

Fichiers créés par la commande Alire init

La commande génère plusieurs fichiers dont le plus intéressant ici se trouve dans tictactoe/src/tictactoe.adb qui correspond à notre programme Ada. Nous obtenons ici une opération introduite par le mot clef **procedure** suivie du nom de notre projet. La première instruction de notre procédure est introduite par le mot clef **begin** qui équivaut à '{' dans beaucoup de langages. Cette procédure définit une seule instruction **null** indiquant l'absence d'intention. Pas très utile, mais c'est un début !

```
procedure Tictactoe is
begin
  null;
end Tictactoe;
```

Pour afficher un texte, nous pouvons utiliser la procédure Put_Line fournie en standard par le paquetage Ada.Text_IO. Comme pour la plupart des autres langages, il faut importer les déclarations en utilisant l'instruction **with** d'Ada. Nous pouvons alors remplacer notre **null** par une instruction plus palpitante :

```
with Ada.Text_IO;
procedure Tictactoe is
begin
  Ada.Text_IO.Put_Line ("Tic-Tac-Toe");
end Tictactoe;
```

Utilisons ensuite la commande alr build pour construire le binaire :

```
cd tictactoe
alr build
```

Le programme exécutable sera produit dans le répertoire bin et aura le même nom que la procédure. Nous pouvons le lancer :

```
$./bin/tictactoe
Tic-Tac-Toe
```

Exécution avec Alire

En général, une fois le projet créé avec Alire, nous allons devoir utiliser des opérations qui ne seront pas forcément disponibles en standard dans le langage. Actuellement, Alire recense plus de 500 paquetages ou bibliotheques permettant de développer des applications en ligne de commande, des applications graphiques, des applications web, mais aussi des applications embarquées pour ARM, RISC-V ou ESP32. Certains de ces paquetages sont des programmes complets qui sont directement utilisables. Si vous cherchez une bibliotheque, la commande alr search sera tres utile, soit pour lister l'ensemble des bibliotheques disponibles (option —list), soit pour rechercher un paquetage spécifique. Une fois la bibliotheque trouvée, il faut l'importer avec la commande alr with. Pour notre exemple, nous allons ajouter un peu de couleur dans les écritures sur le terminal et importer avec Alire la bibliotheque ansiada. Cette bibliotheque propose le paquetage Ada AnsiAda qui gère les séquences d'échappement ANSI.

```
alr with ansiada
```

Pour utiliser cette bibliotheque, il nous suffit d'importer son paquetage AnsiAda et d'utiliser ses opérations. Deux fonctions nous intéressent ici : une première Foreground qui va nous fournir le code ANSI pour la couleur demandée et une seconde Color_Wrap qui nous permettra de changer la couleur et revenir dans le mode précédent. Nous allons définir trois constantes pour les trois couleurs utilisées. En Ada, les chaînes sont représentées par le type String et concaténées avec le symbole '&' (le '+' étant réservé pour tout ce qui est addition).

```
with Ada.Text_IO;
with AnsiAda;
procedure Tictactoe is
  Red   : constant String := AnsiAda.Foreground (AnsiAda.Red);
  Blue  : constant String := AnsiAda.Foreground (AnsiAda.Blue);
  Green : constant String := AnsiAda.Foreground (AnsiAda.Green);
begin
  Ada.Text_IO.Put_Line (AnsiAda.Color_Wrap ("Tic", Red)
            & "-" & AnsiAda.Color_Wrap ("Tac", Blue)
            & "-" & AnsiAda.Color_Wrap ("Toe", Green));
end Tictactoe;
```

À l'exécution, nous obtiendrons :

```
$./bin/tictactoe
Tic-Tac-Toe
$
```

Exécution avec AnsiAda

Un peu d'architecture

La notion de paquetage a été introduite par Ada 83 et reprise ensuite dans la majorité des langages un peu structurés comme Java, Python, Go ou Rust. Dans ces langages, le paquetage permet de regrouper dans une même unité de compilation un ensemble de types, fonctions ou procédures qui ont un lien logique. Ada a toujours séparé la partie déclaration de l'implémentation un peu comme en C où le header .h contient les déclarations et le .c contient le corps des opérations. Cependant, contrairement aux C et C++, Ada 83 est très strict : la partie spécification ne peut contenir que les déclarations. Seules les déclarations visibles dans la partie

spécification peuvent ensuite être utilisées dans les autres paquetages. Un autre aspect intéressant est que la spécification du paquetage Ada peut contenir une partie privée. Cela permet de déclarer des types, des fonctions et des procédures accessibles uniquement aux paquetages fils.
Un paquetage est introduit par le mot clef **package**, nous allons utiliser l'abréviation TTT pour le nom de notre paquetage. La déclaration sera toujours terminée par le mot clef **end** suivie du nom du paquetage afin d'éviter les erreurs. Dans notre exemple, ce paquetage contiendra tous nos types ainsi que les opérations principales contrôlant les règles du jeu *Tic Tac Toe*. La partie privée est introduite par le mot clef **private**.

```
package TTT is
  -- partie publique
private
  -- partie privée
end TTT;
```

Depuis Ada 95, nous pouvons définir un paquetage fils, comme en Java. Ce paquetage, TTT.Display contiendra les opérations d'affichage de notre grille *Tic Tac Toe*.

```
package TTT.Display is
  ...
end TTT.Display;
```

Enfin, nous pouvons aussi déclarer un paquetage fils comme **private** afin de restreindre sa visibilité. Toutes les déclarations de ce paquetage ne seront pas visibles et utilisables en dehors du paquetage parent. Cette restriction est très pratique lorsque l'on analyse ou que l'on doit refactoriser une implémentation puisqu'elle permet de réduire le champ d'analyse. Dans notre exemple, le paquetage TTT.Input va gérer l'entrée utilisateur et son choix dans le jeu.

```
private package TTT.Input is
  ...
end TTT.Input;
```

Le langage n'impose pas de regles sur le nommage et l'organisation des fichiers, mais le compilateur GNAT définit une organisation qu'il est fortement recommandé de suivre. Les spécifications sont définies dans des fichiers avec l'extension .ads *alors que l'implémentation (appelé* **body** *en Ada) doit être écrite dans des fichiers avec l'extension* .adb. *Le nommage des fichiers a aussi son importance : le nom du fichier est basé sur le nom du paquetage ou de la procédure le cas échéant. Dans notre exemple, nous avons l'organisation suivante :*

Figure A

```
tictactoe/
├── alire.toml                  <- définition Alire
├── src
│   ├── tictactoe.adb           <- procedure Tictactoe
│   ├── ttt.adb                 <- package body TTT
│   ├── ttt.ads                 <- package TTT
│   ├── ttt-display.adb         <- package body TTT.Display
│   ├── ttt-display.ads         <- package TTT.Display
│   ├── ttt-input.adb           <- package body TTT.Input
│   └── ttt-input.ads           <- package TTT.Input
└── tictactoe.gpr               <- projet GNAT de compilation
```

Figure A
Les fichiers et leur role

Parlons typage

Une grande force du langage Ada est son système de typage fort. La définition des types d'un programme Ada est une partie importante, car cela va participer à toutes les vérifications qui seront ensuite faites par le compilateur. Ces vérifications font ensuite gagner du temps sur la mise au point et la maintenance. De plus, un bon typage permet de structurer le programme.
Nous souhaitons définir la taille de la grille du jeu. Cette taille est un entier, mais il y a aussi un minimum et un maximum (arbitrairement la grille maximale sera 7x7). Il faut aussi comptabiliser le nombre d'étapes réalisées. Avec une grille de taille N par N, il y a NxN étapes possibles pour remplir toutes les cellules (49 max). Tous ces types ont donc des valeurs possibles différentes. En Ada, un type est introduit par le mot clef **type** et les bornes précisées avec le mot clef **range**. Dans l'exemple, nous trouverons les types suivants :

```
type Counter is new Natural range 0 .. 7;
type Step_Level is new Natural range 0 .. 49;
type Key_Index is new Step_Level;
```

En plus de vérifier les bornes, Ada va vérifier la compatibilité des types entre eux. Une variable de type Key_Index ne pourra pas être assignée ni à une variable de type Step_Level ni Counter, même si les valeurs sont bien représentées par le type cible. Ce sont des espaces de valeur distincts. Cela peut paraître contraignant, mais cela permet d'éviter des erreurs parfois difficiles à trouver dans le cas où l'on s'est trompé de variable dans une affectation ou comparaison. Ada définit la notion de sous-type pour soit représenter un type existant sous un autre nom, soit restreindre les bornes ou certaines valeurs possibles.
Pour le jeu, un système de coordonnées permet d'identifier une cellule dans la grille en partant de 1 jusqu'a la dernière cellule de la grille. Nous allons définir le sous-type Coordinate grâce au mot clef **subtype**. Pour cette définition, nous pouvons utiliser le prédicat 'Last sur le type principal pour obtenir la dernière valeur possible. Pour la taille de la grille, cela n'a pas de sens de démarrer à 1 et il faut au minimum une grille 2x2. On déclare alors le sous-type Board_Size :

```
subtype Coordinate is Counter range 1 .. Counter'Last;
subtype Board_Size is Coordinate range 2 .. Coordinate'Last;
```

Pour représenter chaque cellule du jeu, nous allons utiliser l'énumération Cell avec trois valeurs : Empty, Player_X, Player_0. La valeur Empty doit être exclue pour identifier l'un des deux joueurs que nous pouvons définir avec le sous-type Player :

```
type Cell is (Empty, Player_X, Player_0);
subtype Player is Cell range Player_X .. Player_0;
```

Déclarons ensuite un tableau à deux dimensions pour définir l'ensemble des cellules. Chacune des dimensions utilisera le type Coordinate et les éléments seront des Cell. Le type Board représente cette table à deux dimensions. Cependant, si l'on ne connaît pas ces dimensions, Ada permet de l'indiquer avec la syntaxe '<>'.

```
type Board is array (Coordinate range <>, Coordinate range <>) of Cell;
```

Pour pouvoir identifier une cellule dans la grille, il faut pouvoir préciser la ligne et la colonne. Nous allons définir le type Point représentant une structure ou **record** en Ada. Cette structure se compose de deux valeurs X et Y, chacune du type Coordinate. Il est possible de préciser les valeurs par défaut de ces deux valeurs à la création. Dans notre cas, nous les initialiserons à Coordinate'First représentant la première valeur possible.

```
type Point is record
  X : Coordinate := Coordinate'First;
  Y : Coordinate := Coordinate'First;
end record;
```

Des types de plus en plus complexes

Pour notre exemple, nous allons avoir besoin d'une variable qui définira la taille de la grille ainsi que ses cellules et le nombre d'étapes jouées. Arrangeons-nous pour que la taille de la grille soit définie lorsque nous déclarons notre variable. Dans l'exemple, nous avons une grille 4x4 qui correspond a la déclaration de la variable Game :

```
Game : TTT.Gameplay (4);
```

Le type Gameplay n'a pas besoin d'avoir sa définition visible a l'extérieur du paquetage TTT. Il est déclaré dans la partie privée du paquetage. Autre point intéressant, Ada permet de déclarer un type dont les objets ne pourront pas être copiés : il est possible de déclarer une variable locale ou globale, mais impossible de la copier. C'est le mot clef **limited** qui apporte cette contrainte. Le type Gameplay est déclaré comme suit :

```
package TTT is
  ...
  type Gameplay (Size : Board_Size) is limited private;
private
  type Gameplay (Size : Board_Size) is limited record
    Layout : Board (1 .. Size, 1 .. Size) := [others => [others => Empty]];
    Step   : Step_Level := 0;
  end record;
end TTT;
```

Avec cette déclaration, nous pouvons voir la taille de la grille représentée par Size a l'extérieur du paquetage TTT, mais pas le contenu des cellules représenté par Layout ni le nombre d'étapes jouées représenté par Step. Les valeurs de la grille seront initialisées a Empty lors de la déclaration de notre variable Game grâce a la syntaxe [others => Empty].

Un peu de code

Voyons l'implémentation de quelques opérations intéressantes dans notre jeu comme la fonction Is_Winner qui détermine si un joueur a gagné après avoir choisi une cellule. Cette fonction va renvoyer une valeur booléenne (type Boolean) et recevoir trois paramètres. Pour notre fonction, il suffit de vérifier horizontalement, verticalement ou bien les deux diagonales pour savoir si toutes les cellules sont attribuées au joueur. Avec Ada 2012, nous pouvons utiliser la syntaxe ensembliste **for all** pour vérifier simplement si une condition est réalisée sur un ensemble de cellules. Dans notre cas, la condition sera toujours Game.Layout (<X>, <Y>) = Play pour vérifier le contenu de la cellule. Si la condition est fausse, le **for all** s'arrête et retourne **False**. Le test suivant représenté par **or else** ne sera pas exécuté.

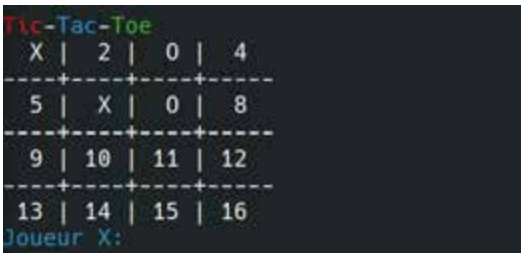
```
package TTT is
  function Is_Winner (Game  : Gameplay;
                      Pos   : Point;
                      Play  : Player) return Boolean is
   (-- Est-ce que le joueur Play a gagné horizontalement ?
    (for all I in 1 .. Game.Size => Game.Layout (Pos.X, I) = Play)
    -- verticalement ?
    or else (for all I in 1 .. Game.Size => Game.Layout (I, Pos.Y) = Play)
    -- sur la diagonale ?
    or else (for all I in 1 .. Game.Size => Game.Layout (I, I) = Play)
    -- sur l'antidiagonale ?
    or else (for all I in 1 .. Game.Size
      => Game.Layout (Game.Size - I + 1, I) = Play));
end TTT;
```

Le lecteur averti aura noté que cette fonction a son implémentation dans la partie spécification, ce qui est interdit par Ada 83, 95, 2005, mais autorisé par Ada 2012 avec l'introduction des *Expression Functions*. De son côté, la procédure Run du paquetage TTT sera écrite dans la partie implémentation du paquetage (précisée par le mot clef **body**). Elle recevra deux paramètres, le premier Game contenant la description et contenu de la grille et le second Winner représentant le résultat. En plus des types des paramètres, Ada oblige a préciser si le paramètre est une entrée de la procédure, une sortie ou les deux. Le paramètre Game de la procédure sera **in out** puisque la procédure aura besoin de le modifier et le résultat Winner est simplement déclaré **out**. Nous commençons par afficher la grille et récupérer le choix du joueur. Nous pouvons immédiatement mettre à jour la cellule de la grille avec le joueur, vérifier s'il a gagné, mettre à jour la grille et passer au joueur suivant. Une boucle **while** assez classique permet de continuer.

```
package body TTT is
  ...
  procedure Run (Game  : in out Gameplay;
                 Winner : out Cell) is
    P : Player := Player_X;
    Choice : Point;
  begin
    Display.Print (Game);
    while not Is_Over (Game) loop
      Input.Get_Input (Game, P, Choice);
      Game.Layout (Choice.X, Choice.Y) := P;
      Game.Step := Game.Step + 1;
      if Is_Winner (Game, Choice, P) then
        Winner := P;
        return;
      end if;
      Display.Print (Game);
      P := Next (P);
```

```
   end loop;
   Winner := Empty;
 end Run;
end TTT;
```

Les procédures Print, Get_Input ne sont pas représentées ici. Après compilation avec la commande alr build et l'exécution du programme tictactoe, voici ce que nous obtenons après avoir joué quelques coups :



Le jeu en action

Des préconditions pour vérifier

En 1986, Bertrand Meyer définit le concept de *Design by Contract* dans son langage Eiffel. Le principe général est que lorsque l'on nomme une procédure ou une fonction, on passe un contrat avec elle. Ce contrat est décrit formellement par des préconditions, des postconditions et des invariants. Les préconditions définissent ce qui doit être vérifié avant d'appeler l'opération tandis que les postconditions indiquent ce que l'opération va garantir après son exécution. Enfin, les invariants précisent ce qui ne change pas.

Ada 2012 a introduit ces préconditions et postconditions afin de permettre de prouver formellement un programme. Elles peuvent etre vérifiées a l'exécution du programme ou utilisées comme support de preuve statique avec gnatprove comme nous le verrons plus loin. Lorsque les préconditions et postconditions sont activées a l'exécution (avec l'option de compilation -gnata), si elles ne sont pas vérifiées, une exception avec un message est levée.

Ajoutons une précondition a notre fonction Is_Empty qui vérifie si une cellule du jeu est vide ou déjà jouée. Nous donnons une position, mais cette position doit bien sûr être dans notre grille. Nous exprimons la précondition dans la spécification avec l'aspect Pre de Ada 2012 en indiquant l'expression booléenne qui doit être vérifiée. Dans notre cas, les valeurs X et Y ne doivent pas dépasser la taille de la grille. Il n'est pas utile de vérifier la borne inférieure, car le type Board_Size le fait déjà.

```
function Is_Empty (Game : in Gameplay;
                   Pos  : in Point) return Boolean is
 (Game.Layout (Pos.X, Pos.Y) = Empty)
  with Pre => Pos.X <= Game.Size and then Pos.Y <= Game.Size;
```

Une précondition peut être assez complexe et il est souvent utile d'avoir recours à des fonctions définies uniquement pour les vérifier. Dans notre exemple, nous avons la procédure Run qui se charge de gérer le jeu. Quand nous la lançons, il faut que la grille soit vide. Nous pouvons donc exprimer cela simplement :

```
procedure Run (Game  : in out Gameplay;
               Winner : out Cell)
 with Pre => Is_Empty (Game);
```

Reste maintenant à implémenter cette fonction Is_Empty avec la syntaxe ensembliste **for all** on peut l'écrire simplement :

```
function Is_Empty (Game : in Gameplay) return Boolean is
 (for all Row in 1 .. Game.Size =>
   (for all Col in 1 .. Game.Size =>
     Game.Layout (Row, Col) = Empty));
```

Les postconditions sont précisées via l'aspect Post. Dans le cas de notre procédure Get_Input renvoyant une position sélectionnée par le joueur, nous nous attendons à ce que cette position indique bien une cellule du jeu (c'est le test Is_Valid), mais aussi que la cellule correspondante soit vide (c'est le test Is_Empty).

```
private package TTT.Input is
  procedure Get_Input (Game  : in Gameplay;
                       From  : in Player;
                       Result : out Point) with
   Pre => not Is_Over (Game),
   Post => Is_Valid (Game, Result) and then Is_Empty (Game, Result);
end TTT.Input;
```

Un peu de SPARK

Prouver formellement un programme est complexe et bien souvent le programme seul ne suffit pas. Bernard Carré et Trevor Jennings ont créé SPARK à partir d'Ada 83 dans les années 1990. Le langage SPARK est un sous ensemble d'Ada où certaines fonctionnalités sont interdites, car impossibles à prouver. Au départ, les pré et postconditions étaient exprimées sous forme de commentaires, mais depuis Ada 2012 et sa version SPARK 2014, elles font maintenant directement partie du langage.

Le mode SPARK est activé grâce à une directive SPARK_Mode dans la déclaration du paquetage et elle ne s'applique qu'au paquetage concerné. L'activation de cette directive permet d'utiliser gnatprove et d'accéder à la preuve de programme.

```
package TTT with SPARK_Mode => On is
...
end TTT;
```

L'outil gnatprove ne va prouver que les paquetages ayant l'aspect SPARK_Mode positionnés à On. Il est possible de ne prouver qu'une partie et de désactiver la vérification pour certaines opérations uniquement. gnatprove va utiliser plusieurs assistants de preuve comme Alt-Ergo, Colibri, CVC5, Z3. D'abord, installons gnatprove avec Alire et la commande suivante :

```
alr get gnatprove
```

Pour lancer l'assistant de preuve, il faut l'avoir ajouté dans son PATH et de lancer la commande via alr tout simplement :

```
alr gnatprove
```

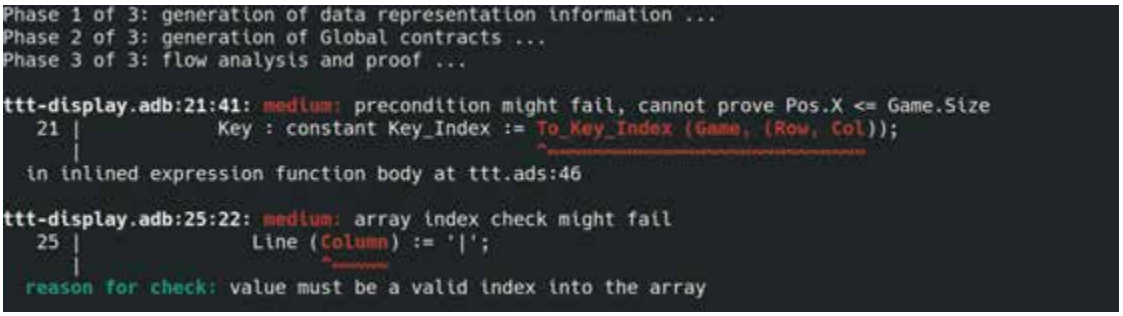


Figure B

Essai de preuve

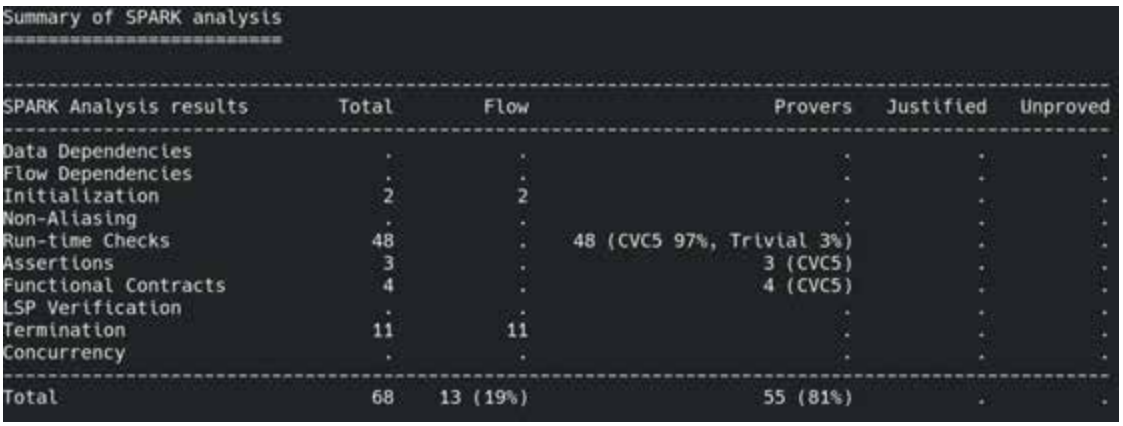


Figure C

Le jeu prouvé

Parfois l'assistant de preuve n'a pas assez d'informations pour prouver que les valeurs de certaines variables ne seront pas en dehors des limites. C'est le cas pour nous malgré les pré et postconditions, et gnatprove indique plusieurs erreurs. En particulier, l'assistant de preuve ne peut pas prouver que l'accès a la chaîne Line de la procédure Print sera toujours dans les limites. **Figure B**

Dans notre exemple, il suffit d'ajouter une assertion pour indiquer que la variable Column ne dépassera jamais une certaine limite. En plus, il nous faut préciser un Loop_Invariant c'est-à-dire une autre condition qui sera toujours vérifiée, quelle que soit l'itération de la boucle.

```
package body TTT.Display is
  procedure Print (Game : in Gameplay;
                   Row  : in Coordinate) is
    ...
  begin
    for Col in 1 .. Game.Size loop
      pragma Loop_Invariant (Column = (Natural (Col) - 1) * 5);
      pragma Assert (Column <= Line'Last - 5);
      ...
    end loop;
    ...
  end Print;
  ...
end TTT.Display;
```

Lorsque tout est prouvé, le résumé de gnatprove indique qu'il n'y a plus d'élément Unproved. À ce stade, nous avons prouvé qu'il n'y aura pas d'erreur Run-time checks, c'est-à-dire que toutes les valeurs seront bien dans les bornes définies par leur type, que les assertions et contrats des fonctions seront bien vérifiés et que toutes les boucles auront une fin et le programme s'achèvera (Termination). **Figure C**

Un langage en constante évolution

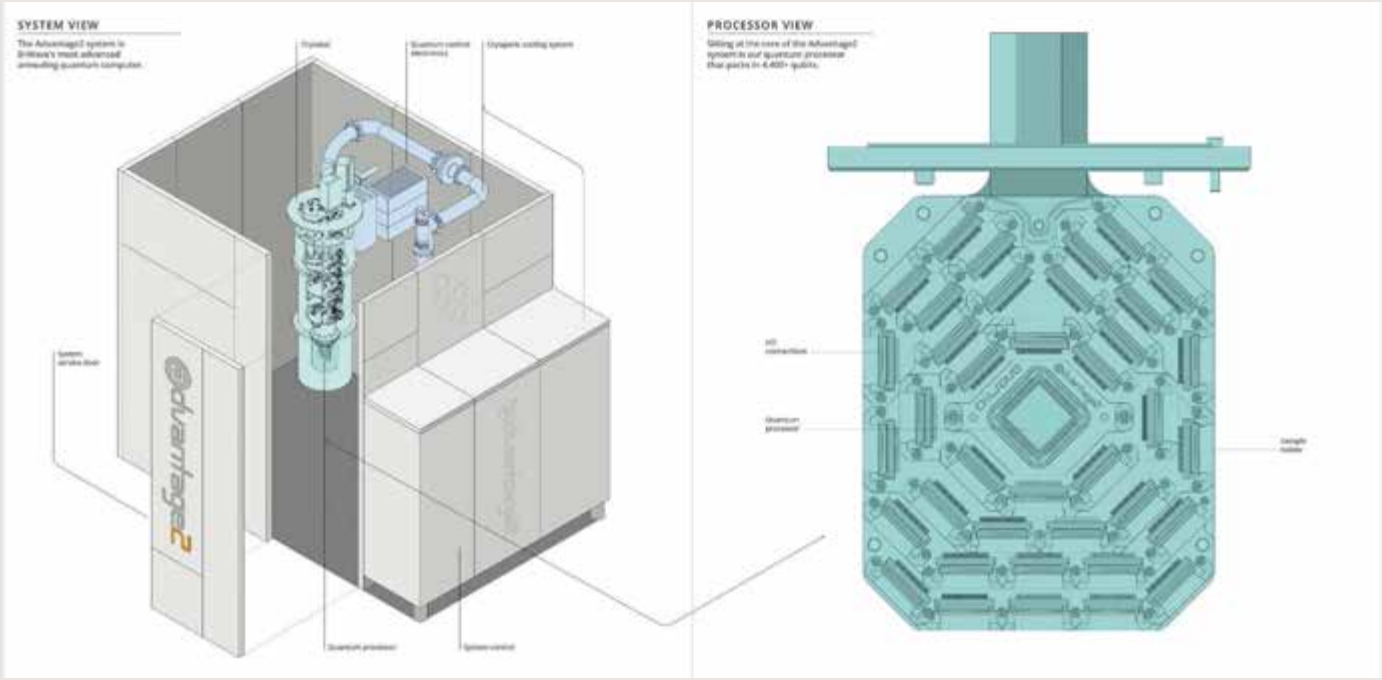
Le langage continue d'évoluer et le *Ada Rapporteur Group* (ARG) en charge de l'évolution et de la normalisation du langage vient de lancer un appel aux commentaires sur leur projet GitHub https://github.com/Ada-Rapporteur-Group/User-Community-Input.

Pour en apprendre plus sur Ada, le site https://learn.adacore.com/ est excellent pour apprendre en ligne tout en essayant via son navigateur. Installer Alire n'est pas nécessaire pour tester sa compréhension du langage. De plus, à l'heure des IA génératives, il devient maintenant plus facile d'apprendre ce langage et d'implémenter des programmes avec ChatGPT.

La communauté Ada est particulièrement active sur le forum https://ada-lang.io/ où vous pouvez poser des questions (en anglais). Ce groupe organise une fois par mois un « Meetup » où les participants présentent leurs projets et donnent des informations. En parallèle des Ada Dev room au FOSDEM, une journée *Ada Developers Workshop* est organisée chaque année après la conférence *Ada Europe*.

Informatique quantique

PARTIE 1



© D-Wave

Programmez! vous propose un grand dossier sur l'informatique quantique en 2 parties : dans ce numéro d'été et dans le numéro de septembre. Notre dernier hors-série sur le sujet remonte à 2 ans. Depuis, il y a eu de nombreuses annonces.

Il y a une affirmation qui revient régulièrement : la suprématie quantique. Ce concept a été créé par John Preskill en 2012. Pour simplifier, la suprématie quantique sera atteinte quand un ordinateur quantique pourra faire aussi bien qu'un ordinateur « classique » dans tous les domaines. Bref, quand le quantique surpassera l'ordinateur et pourra le faire dans un temps raisonnable.

Google avait annoncé la suprématie quantique en 2019. D-Wave l'avait affirmé en mars dernier. Dans le même temps, les ordinateurs continuent à évoluer grâce à de nouvelles architectures, des composants toujours plus spécialisés. Mais, nous pouvons nous interroger : faut-il réellement s'obstiner à comparer l'ordinateur quantique et l'ordinateur ?

Car fondamentalement, un ordinateur quantique est différent. Il n'est pas question de pouvoir en acheter un. Il n'a pas vocation à être une machine de gaming ou pour de la bureautique.

Comme nous le verrons dans ce double dossier, une des questions cruciales est : faut-il concevoir un ordinateur quantique universel ou spécialisé ? Et il existe plusieurs technologies pour concevoir des processeurs quantiques (les QPU), tout comme, il existe plusieurs modèles de programmation.

Un autre frein à l'ordinateur quantique est la capacité des QPU à corriger les erreurs et à supprimer les bruits, les parasites. C'est aujourd'hui un des enjeux les plus importants : un ordinateur quantique à correction d'erreurs. Ces erreurs viennent de l'instabilité des états au niveau du qubit. L'objectif des principaux constructeurs est de construire un ordinateur quantique sans erreurs d'ici 2030, les plus optimistes l'estiment à 2028.

Pour poursuivre ce dossier, rendez-vous le 9 octobre pour notre conférence informatique quantique au campus de 42 Paris. Tous les détails sur programmez.com

Bon été quantique. Et révisez votre chat de Schrödinger.

François Tonic

Informatique quantique : Comprendre l'émulation quantique avec Eviden Qaptiva

Le calcul quantique vise, à terme, à développer des « ordinateurs » dont le fonctionnement repose sur les principes fondamentaux de la mécanique quantique, en rupture avec le modèle binaire des ordinateurs classiques.

Dans des conditions particulières, le qubit (quantum bit ou bit quantique) possède deux propriétés essentielles issues de la mécanique quantique : la **superposition** et **l'intrication**. Contrairement au bit classique, qui ne peut prendre que les valeurs 0 ou 1, le qubit peut exister simultanément dans plusieurs états, ouvrant la voie à des capacités de calcul inédites. Les gains de temps de l'informatique quantique ne se mesurent pas en heures ou jours, Il s'agit potentiellement de résoudre, en quelques minutes seulement, des problèmes complexes que les ordinateurs les plus puissants d'aujourd'hui mettraient des dizaines, voire des centaines d'années à traiter — ouvrant ainsi la voie à la résolution de problématiques jusqu'ici inaccessibles.

Cependant, le développement et la mise au point de ces futurs ordinateurs restent extrêmement complexes, et de nombreux défis doivent encore être relevés avant qu'ils ne deviennent des systèmes véritablement utilisables pour résoudre des problèmes concrets. À ce jour, il est généralement admis que l'horizon de maturité pour ces systèmes, dits « FTQC » (Fault-Tolerant Quantum Computers), se situe entre 10 et 15 ans.

L'émulation quantique constitue aujourd'hui une excellente porte d'entrée dans le domaine de l'informatique quantique, car elle est déjà accessible.

Cette technologie permet aux utilisateurs de simuler le comportement d'un ordinateur quantique, offrant ainsi une ressource précieuse utile pour la programmation, l'optimisation, la compilation et l'émulation de code.

Grâce à l'émulation quantique, les développeurs et chercheurs peuvent commencer à travailler avec les technologies quantiques pour résoudre des problèmes complexes, sans être limités par la disponibilité des QPU (Quantum Processing Units), les coûts d'accès ou la puissance de calcul.

Les émulateurs quantiques reposent sur du matériel classique associé à un environnement logiciel spécifique. L'émulation quantique consiste à développer et affiner des algorithmes quantiques sur des ordinateurs classiques, en vue de leur exécution future sur des processeurs quantiques (QPU).

Eviden Qaptiva : de la conception à l'exécution d'algorithmes quantiques, sans dépendance matérielle

La maturité actuelle de l'informatique quantique ne permet pas encore de traiter des problèmes concrets.

Nous sommes toujours dans l'ère dite « NISQ » (Noisy Intermediate-Scale Quantum), caractérisée par des machines limitées et sujettes au bruit. De nombreux défis restent à relever, notamment en ce qui concerne la qualité des qubits, le passage à l'échelle, la correction d'erreurs quantiques, etc.

C'est pourquoi, dès 2016, Atos a lancé le programme « Atos Quantum Program », articulé autour d'une offre matérielle et logicielle d'émulation quantique : la Quantum Learning Machine (QLM). Cette solution a depuis évolué pour devenir **Qaptiva**, une plateforme complète dédiée à l'exploration et au développement du calcul quantique.

Qaptiva repose sur une ambition claire : aider les équipes — qu'elles soient issues du secteur privé, public, académique ou de l'écosystème start-up — à :

- Se former,
- Identifier des cas d'usage,
- Développer des algorithmes et des applications,
- Expérimenter dans un environnement maîtrisé.

Le tout, en anticipant la disponibilité du véritable matériel quantique sans erreur (*Fault-Tolerant Quantum Computers – FTQC*), et en offrant ainsi à la communauté scientifique l'opportunité de se préparer dès aujourd'hui aux défis technologiques de demain.

Grâce à son infrastructure matérielle dédiée, Qaptiva émule l'exécution d'un programme comme le ferait un véritable ordinateur quantique (QPU). Les applications peuvent y être développées et testées à l'aide d'outils spécifiques, soit directement sur l'émulateur, soit sur un ordinateur quantique NISQ existant.

Qaptiva est une appliance d'émulation quantique reposant sur une architecture matérielle classique (processeur x86 Intel) et un environnement logiciel dédié. Indépendante des technologies de qubits, elle adopte une approche « hardware-agnostic » permettant de développer, valider et tester des algorithmes sur tout type d'ordinateur quantique. Pour rappel il existe plusieurs moyens de concevoir un qubit physique. Aujourd'hui les technologies les plus utilisées reposent sur des atomes, des ions piégés, ou des spins d'électron. On peut également concevoir un qubit supraconducteur (transmon). Enfin, une autre particule possible est le photon, en jouant sur ses polarités. En France les représentants de ces technologies sont Pasqal (atomes froids), Quandela (photons), A&B (supraconducteur), Quobly et C12 (spin d'électron). Mais aucune de ces technologies n'est « parfaite » et chacune possède ses avantages et ses inconvénients.

Une nouvelle offre est désormais disponible sous le nom de Qaptiva HPC. Elle permet d'émuler des circuits quantiques

Damien Nicolazic
Eviden
Quantum Computing Consultant

Grégory Vieux
Eviden
Quantum Computing Consultant

Mariem Bahri
Eviden
Quantum Computing Consultant

Olivier Hess
Eviden
Global Quantum Computing Technology Advisor
Quantum Computing Leader France

Robert Wang
Eviden
Quantum Computing Consultant

sur des clusters HPC, offrant ainsi une capacité de simulation à grande échelle pour tester et valider des algorithmes dans un environnement haute performance.

Qaptiva intègre un environnement complet de programmation quantique, avec les fonctionnalités suivantes :

- Modèle de programmation universel, indépendant du matériel et des logiciels.
- Langage hybride de haut niveau, basé sur Python, adapté aux algorithmes NISQ, notamment variationnels.
- Prise en charge du calcul par portes, du recuit quantique et du calcul analogique, avec plugins multi-langages et SDK intégré.
- Couche d'abstraction (Atos Quantum ASseMbly Python) permettant de générer du code AQASM et de simuler de manière hybride.
- Création facilitée de portes et de sous-programmes réutilisables.
- Suite d'optimisation pour adapter les circuits à différents matériels via des plugins personnalisés.
- Module d'émulation du bruit, sans réduction du nombre de qubits disponibles.

Figures 1 et 2

Présentation des émulateurs intégrés dans Qaptiva

La programmation quantique s'invite peu à peu dans le quotidien des développeurs : bibliothèques, environnements de développement et même machines accessibles en ligne permettent désormais d'écrire et d'exécuter du code quantique.

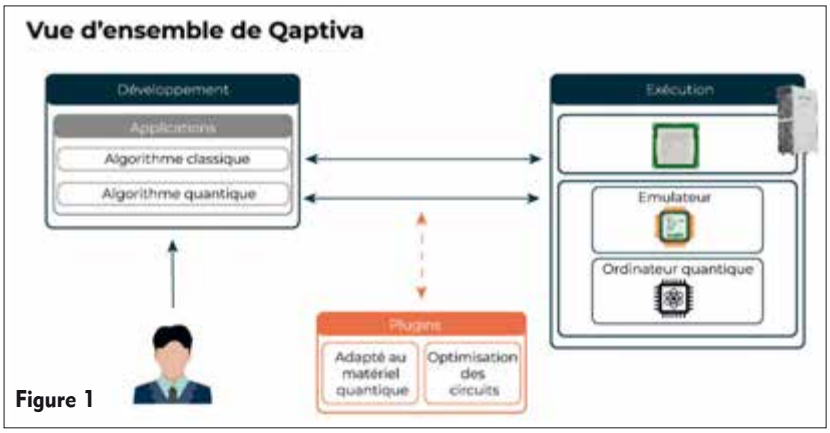


Figure 1

Figure 2

Mais programmer pour un processeur quantique ne consiste pas simplement à transposer du code classique sur une nouvelle architecture. Il s'agit d'un véritable changement de paradigme, car l'information y est traitée de manière fondamentalement différente.

Qaptiva intègre une suite d'émulateurs quantiques conçus pour répondre aux différents besoins des développeurs, chercheurs et ingénieurs quantiques. Ces émulateurs permettent de simuler le comportement d'un ordinateur quantique sur une infrastructure classique, tout en offrant un environnement flexible, évolutif et indépendant du matériel quantique sous-jacent.

Chaque émulateur a été conçu pour répondre à un usage bien précis, du prototypage rapide d'algorithmes à la simulation fine intégrant du bruit, en passant par l'optimisation de circuits pour des architectures matérielles ciblées. Cette diversité offre aux utilisateurs un environnement maîtrisé pour tester, comparer et affiner leurs approches, sans dépendre des contraintes ou de la disponibilité des QPU réels.

Dans cette section, nous vous présentons les principaux émulateurs disponibles dans Qaptiva, leurs caractéristiques techniques, ainsi que les cas d'usage auxquels ils répondent.

Programmation quantique : émulateurs de QPU à portes

Le modèle de programmation le plus répandu aujourd'hui est celui des circuits **quantiques à portes logiques**. Dans cette approche, un programme est représenté comme une séquence d'opérations élémentaires — les portes quantiques — appliquées à des qubits, unités de base de l'information quantique, analogues aux bits classiques.

Ce modèle présente un double avantage :

- D'une part, il est structuré comme les circuits logiques classiques que tout développeur connaît — avec des portes (XOR, NOT, etc.), une chronologie d'exécution, des entrées et des sorties.
- D'autre part, il permet d'exploiter les propriétés typiquement quantiques que sont la superposition, l'intrication et le caractère probabiliste de la mesure, qui confèrent aux algorithmes quantiques leur puissance potentielle.

Cela peut surprendre : pourquoi code-t-on encore au niveau des portes ? En informatique classique, cela évoquerait le codage en assembleur — un niveau de détail que les langages de haut niveau ont depuis longtemps abstrait.

En informatique quantique, cette abstraction est encore en construction. Certes, des langages plus évolués comme Q#, Quipper ou Silq ont vu le jour, mais dans la pratique, la majorité des programmes doivent encore être traduits en séquences de portes élémentaires pour être exécutés sur les machines universelles actuelles, fondées sur un modèle discret et numérique, ne reconnaissant que ce format.

Autrement dit, programmer avec des portes revient aujourd'hui à écrire du code bas niveau pour le quantique : ce n'est pas toujours idéal pour la productivité, mais c'est indispensable pour comprendre les mécanismes, tester des algorithmes et exploiter les émulateurs ou processeurs existants.

C'est aussi le format standard pris en charge par la majorité des plateformes actuelles, qu'il s'agisse d'outils open source (Qiskit, Cirq…) ou de solutions industrielles comme myQLM/Qaptiva.

Les portes quantiques : briques de base des circuits

Quel que soit le support technologique utilisé, la programmation quantique repose sur un ensemble réduit mais fondamental de portes logiques [1]. Ces portes quantiques manipulent les qubits, unités d'information dont l'état peut être une superposition des états de base $|0\rangle$ et $|1\rangle$, représentant simultanément 0 et 1 jusqu'à ce qu'une mesure l'effondre vers l'un de ces deux états.

À la différence des portes logiques classiques, les portes quantiques sont modélisées par des opérateurs unitaires, c'est-à-dire des matrices complexes qui conservent les probabilités et assurent la réversibilité des transformations. Elles peuvent être vues géométriquement comme des rotations dans un espace vectoriel complexe.

Mathématiquement, un qubit est représenté par un vecteur colonne de dimension 2, et une porte par une matrice 2×2 (ou plus, selon le nombre de qubits concernés). Parmi les portes les plus courantes, et qui seront utilisées dans la section suivante, figure la porte **X**, équivalente à une inversion logique (NOT), qui échange les états $|0\rangle$ et $|1\rangle$:

$$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \quad X\begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}.$$

La porte Hadamard, **H**, permet de créer une superposition équilibrée à partir d'un état de base. Elle est définie par la matrice :

$$H = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}, \quad H\begin{pmatrix} 1 \\ 0 \end{pmatrix} = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

Enfin, la porte **CNOT** (Controlled-NOT) est un exemple de porte agissant sur deux qubits : elle laisse le second (qubit cible) inchangé si le premier (qubit de contrôle) est dans l'état $|0\rangle$, mais applique une porte X au qubit cible si le qubit de contrôle est dans l'état $|1\rangle$. Elle est représentée par la matrice 4×4 suivante :

$$\text{CNOT} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

Cette porte est notamment essentielle pour créer des états intriqués, exploités dans de nombreux algorithmes quantiques.

Exemple d'implémentation de l'algorithme de Bernstein-Vazirani : retrouver un mot caché en une seule requête

Dans cette section, des techniques de programmation quantique fondées sur le modèle des portes logiques sont mises en œuvre pour implémenter l'algorithme de Bernstein-Vazirani à l'aide de la suite quantique myQLM/Qaptiva, développée par Eviden. L'exécution s'effectue sur un émulateur idéal, c'est-à-dire en négligeant les effets de bruit et les erreurs matérielles. Deux approches complémentaires sont présentées : une implémentation classique reposant sur l'algèbre linéaire, et une représentation par Matrix Product State (MPS) [2].

L'algorithme de Bernstein-Vazirani [3] est un exemple emblématique d'un avantage quantique obtenu grâce à une meilleure exploitation de la structure mathématique d'un problème. Bien que simple dans sa formulation, il illustre de manière efficace comment les principes de superposition et d'interférence permettent de surpasser certaines limitations classiques.

L'énoncé du problème est le suivant : on considère la fonction booléenne $f : \{0,1\}^n \to \{0,1\}$ (mod 2) définie par $f(\vec{x}) = \vec{a} \cdot \vec{x} \pmod 2$, où $\vec{x} \in \{0,1\}^n$ est un vecteur d'entrée choisi par l'utilisateur, $\vec{a} \in \{0,1\}^n$ est un vecteur fixe et inconnu, le produit scalaire $\vec{a} \cdot \vec{x}$ est calculé bit à bit, suivi d'une réduction modulo 2. La fonction retourne donc un bit, déterminé par la parité du nombre de positions où $\vec{a}$ et $\vec{x}$ ont des bits à 1 simultanément.

L'objectif est de retrouver l'intégralité du vecteur $\vec{a}$ en interrogeant f. Sur un ordinateur classique, il est nécessaire d'effectuer n évaluations de f, en testant les vecteurs de base, afin d'isoler chaque bit de $\vec{a}$. L'approche quantique permet de résoudre ce problème en une seule évaluation de f, en tirant parti de la **superposition** pour interroger simultanément toutes les valeurs de $\vec{x}$.

L'explication complète de l'algorithme dépasse le cadre de cet article, mais ses grandes étapes peuvent être résumées simplement.

On commence par préparer un registre de qubits dans l'état $|0\rangle^{\otimes n}$, puis on applique une transformation de Hadamard à chaque qubit, de manière à créer une superposition uniforme de tous les états possibles. Le système représente alors simultanément toutes les entrées de la fonction que l'on cherche à interroger. Une opération quantique appelée *oracle* U_f est ensuite appliquée. Cet oracle agit comme une « boîte noire » : il encode une fonction sans en révéler le contenu, mais permet d'y accéder via une transformation quantique. Concrètement, il modifie la phase de chaque terme de la superposition en fonction du résultat de la fonction sur cet état. Cette étape, bien que non observable directement, prépare les conditions pour que l'interférence puisse révéler l'information recherchée.

Enfin, une seconde application de portes de Hadamard permet de faire interférer les amplitudes des différents états. Par un phénomène d'interférences constructives et destructives, seul l'état correspondant à la chaîne recherchée, $\vec{a}$, reste avec une probabilité maximale à la mesure. Ainsi, le résultat de l'algorithme donne directement la réponse en une seule interrogation de la fonction. Mathématiquement, l'implémentation peut être décrite de la façon suivante :

$$|0\rangle^n \xrightarrow{H^{\otimes n}} \frac{1}{\sqrt{2^n}} \sum_{x \in \{0,1\}^n} |x\rangle \xrightarrow{U_f} \frac{1}{\sqrt{2^n}} \sum_{x \in \{0,1\}^n} (-1)^{f(x)} |x\rangle \xrightarrow{H^{\otimes n}} \frac{1}{2^n} \sum_{x,y \in \{0,1\}^n} (-1)^{f(x)+x \cdot y} |y\rangle = |a\rangle.$$

Pour illustrer concrètement l'algorithme de Bernstein-Vazirani, celui-ci est codé en Python à l'aide de myQLM/Qaptiva, une plateforme complète de programmation quantique accessible via une interface Python. Le code peut être exécuté dans un environnement interactif tel que Jupyter Notebook.

Implémentation avec le solveur PyLinalg

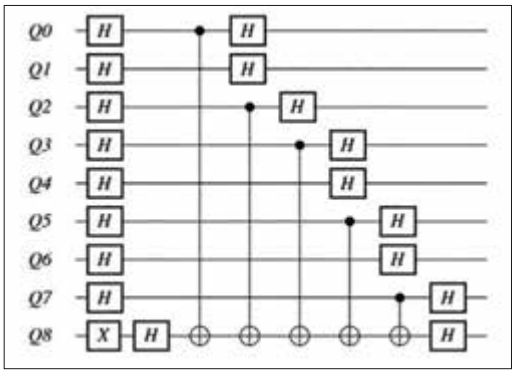
Le solveur PyLinalg repose sur une simulation exacte par algèbre linéaire, en représentant l'état quantique sous forme de vecteur d'état dense de taille 2^n, où n est le nombre de qubits du circuit. L'émulateur manipule directement l'état $|\psi\rangle \in C^{2^n}$, et applique les portes quantiques comme des matrices unitaires agissant sur cet espace. Cela permet une

simulation fidèle et déterministe de l'algorithme, sans approximation ni bruit. Cette approche est particulièrement adaptée aux circuits de petite à moyenne taille, car la mémoire nécessaire croît exponentiellement avec n. Sa précision en fait un outil idéal pour le prototypage rapide, le débogage de circuits et l'exploration pédagogique d'algorithmes.

Voici une implémentation simple et complète du circuit pour une chaîne secrète de 8 bits :

```
# Import des briques nécessaires :
# - Program : permet de construire un circuit quantique de manière structurée
# - H, CNOT, X : portes quantiques de base (Hadamard, CNOT, Pauli-X)
# - PyLinalg : solveur exact basé sur l'algèbre linéaire, utilisé pour simuler
#              l'exécution du circuit
from qat.lang import Program, H, CNOT, X
from qat.qpus import PyLinalg

# == Initialisation ==
# Taille du registre (8 bits)
n = 8

# Le vecteur binaire secret à découvrir par l'algorithme, donné sous forme de chaîne
# de caractères
secret_string = "10110101"

# Initialisation d'un nouveau programme quantique
prog = Program()

# Allocation de n+1 qubits :
# - les n premiers serviront à encoder l'entrée x
# - le dernier est le qubit auxiliaire utilisé pour l'évaluation de la fonction f(x)
qbits = prog.qalloc(n + 1)

# Préparation de l'oracle : mettre le dernier qubit à |1⟩ via une porte X (NOT)
prog.apply(X, qbits[n])

# == Construction de l'algorithme ==
# Etape 1 : appliquer Hadamard à tous les qubits
for qubit in qbits:
    prog.apply(H, qubit)

# Etape 2 : oracle d'évaluation de f(x) = a·x mod 2
# Cela applique une porte CNOT contrôlée sur chaque bit i de x si a_i = 1
for i in range(n):
    if secret_string[i] == "1":
        prog.apply(CNOT, qbits[i], qbits[n])

# Etape 3 : Appliquer Hadamard à nouveau sur tous les qubits
for qubit in qbits:
    prog.apply(H, qubit)

# == Compilation et exécution ==
# Compilation du Program en Circuit, structure finale statique visualisable
circuit = prog.to_circ()
circuit.display()

# Soumission
job = circuit.to_job()
result = PyLinalg().submit(job)

# Affichage du résultat
for sample in result:
    print(
        f"Chaîne binaire trouvée : {sample.state} "
        + f"avec probabilité {sample.probability:.2f}"
    )
```

Le résultat suivant est obtenu.



Chaîne binaire trouvée : |101101011> avec probabilité 1.00

Il est possible de visualiser le circuit quantique, avec le détail des portes appliquées à chaque qubit. Après exécution de l'algorithme de Bernstein-Vazirani, le mot caché est retrouvé avec une probabilité de 100 %, comme attendu théoriquement. Le vecteur d'état retourné par la simulation inclut également le qubit auxiliaire utilisé par l'oracle. Il faut donc veiller à extraire uniquement les n premiers qubits pour récupérer correctement le mot caché.

Implémentation avec le solveur MPS

Le solveur MPS (Matrix Product State) repose sur une représentation plus compacte de l'état quantique que la méthode dense utilisée par Linalg. Au lieu de manipuler un vecteur d'état de taille 2^n, MPS découpe ce vecteur en une chaîne de blocs plus petits (des tenseurs), ce qui permet de simuler plus efficacement les circuits, tant que l'intrication entre qubits reste limitée.

Cette approche permet de simuler des circuits avec un grand nombre de qubits, là où les méthodes classiques deviennent rapidement trop coûteuses en mémoire.

Le solveur MPS est donc particulièrement bien adapté aux circuits peu profonds ou à structure linéaire, et constitue une solution intéressante pour tester des algorithmes à plus grande échelle tout en conservant une bonne précision. Toutefois, cette méthode n'autorise que les portes agissant sur deux qubits voisins. Pour satisfaire cette contrainte topologique, il est nécessaire de compiler le circuit en une forme compatible en utilisant un compilateur.

Pour utiliser l'émulation basée sur MPS, il suffit de reprendre le code précédent en important le simulateur concerné ainsi

que le compilateur Nnizer, qui adapte le circuit aux contraintes de cette méthode.

```
# Import de l'émulateur MPS
from qat.qpus import MPS
# Import du compileur Nnizer
from qat.plugins import Nnizer
```

Il suffit ensuite de soumettre le circuit défini précédemment à notre émulateur, composé de l'objet MPS et du compilateur Nnizer.

```
# Soumission
job = circuit.to_job()
emulatedQPU_stack = Nnizer() | MPS()
result = emulatedQPU_stack.submit(job)
```

Programmation quantique : émulateurs de QPU analogiques

Un QPU analogique est un type de processeur quantique qui utilise des méthodes analogiques pour effectuer des calculs. Contrairement aux QPU à portes quantiques, les QPU analogiques exploitent les interactions naturelles entre atomes ou particules pour résoudre des problèmes spécifiques. On distingue deux types de QPU analogiques : **les machines à recuit quantique et les machines dites "analogiques" ou "simulateurs quantiques"**. Comme on le voit, la terminologie prête à confusion, et le débat reste ouvert.

Différences entre les deux types de QPU analogiques

En physique, l'Hamiltonien représente l'ensemble de l'énergie d'un système. La différence fondamentale entre les deux types réside dans la nature de l'Hamiltonien utilisé et dans son évolution temporelle.

Le recuit quantique repose sur un Hamiltonien de type Ising [4] et s'appuie sur le théorème adiabatique qui sera expliqué dans le paragraphe 3.2.2. En revanche, l'autre type de QPU **analogique** peut implémenter des Hamiltoniens non Ising et ne nécessite pas nécessairement une évolution adiabatique.

Sur l'aspect d'évolution, l'Hamiltonien du recuit quantique, $H(t)$, change dans le temps de manière contrôlée (adiabatique). C'est une interpolation entre t_0 et t_1.

$$H(t) = A(t)H(t_0) + B(t)H(t_1)$$

avec $A(t_0) \gg 0, B(t_0) = 0;\ A(t_1) = 0, B(t_1) \gg 0$

Dans le cas d'un QPU analogique hors recuit, l'Hamiltonien H peut être dépendant ou indépendant du temps, selon le phénomène à simuler. Dans le cas où H est fixe, on peut appliquer un couplage (par exemple), puis on laisse le système évoluer pendant un temps t. A la fin, on mesure l'état, ce qui permet d'accéder aux propriétés physiques simulées.

$$|\psi(t)\rangle = e^{-iHt/\hbar}|\psi(0)\rangle$$

Ici, le temps t est un paramètre expérimental : il n'y a pas d'interpolation.

En raison de cette différence fondamentale, le recuit quantique est principalement utilisé pour les problèmes d'optimisation, tandis que le QPU analogique hors recuit peut également servir à la simulation de systèmes physiques, comme des molécules chimiques.

Quel que soit le type, un QPU analogique n'est généralement pas utilisé pour des calculs plus généraux, comme les algorithmes de HHL, Grover ou Shor. Ces algorithmes nécessitent le plus souvent des QPU à portes, capables de manipuler les qubits de manière plus flexible et précise.

Les deux types de QPU analogiques — le recuit quantique et les machines analogiques hors recuit — ainsi que leurs émulations, seront présentés dans les sections suivantes.

Le recuit quantique

Le recuit quantique est un paradigme de programmation quantique qui tire son appellation du recuit. Le recuit est une technique en métallurgie qui vise à modifier les propriétés d'un métal. C'est un traitement thermique qui se fait en chauffant un métal au-dessus d'une certaine température appelé température critique du métal pour ensuite le refroidir lentement. Ce refroidissement lent et contrôlé est appelé recuit. Le recuit du métal améliore en fait sa ténacité.

Le but de cette technique est, par exemple, d'obtenir un métal de haute qualité, aux propriétés améliorées. Elle s'inscrit pleinement dans une logique d'optimisation que l'on retrouve aussi bien dans le monde physique et la nature que dans le domaine informatique. En effet, en refroidissant, le métal se réorganise en une configuration plus stable et de ce fait on obtient un faible niveau d'énergie. On souhaite donc que la forme finale ait une faible énergie. En ce sens, on se trouve face à un problème de minimisation d'énergie du matériau.

Le recuit est donc un processus physique d'optimisation, et c'est exactement l'inspiration derrière le recuit simulé.

Le **simulated annealing** ou **recuit simulé** est une méthode métaheuristique d'optimisation développée en 1983. Une des plus grandes difficultés d'un problème d'optimisation c'est d'être bloqué dans un minimum local. Pour pallier cela, le recuit simulé effectue une recherche aléatoire dans l'espace des solutions.

Si une modification améliore la valeur de la fonction de coût, elle est toujours acceptée. Le point clé est que si une modification détériore la valeur de la fonction de coût, elle peut tout de même être acceptée, avec une certaine probabilité qui dépend de la « température » selon un critère appelé critère de Metropolis :

- À haute température, le système est en exploration et même les mauvaises modifications ont une grande probabilité d'être acceptées.
- À mesure que la température diminue progressivement selon le calendrier de recuit, le système devient plus sélectif, n'acceptant à terme que les meilleures modifications.

Mathématiquement, la décision d'accepter une modification moins "bien" s'appuie sur des concepts de physique statistique, notamment les méthodes de Monte-Carlo par chaîne de Markov (MCMC), où on construit une marche aléatoire à travers l'espace des solutions où chaque étape dépend uniquement de l'état actuel, cette « absence de mémoire » imite la façon dont les systèmes physiques explorent les configurations.

Maintenant, il est temps d'introduire le **recuit quantique,** ou quantum annealing.

Le recuit quantique peut être vu comme une généralisation du recuit. En recuit quantique, un phénomène appelé l'effet tunnel permet au système de traverser des barrières d'énergie sans avoir besoin de passer par-dessus, comme s'il prenait un

tunnel dans cette barrière, comme le nom l'indique. Cela peut être particulièrement utile face à des paysages de solutions avec des vallées étroites et profondes. Dans le recuit quantique simulé (SQA : **simulated quantum annealing)**, on remplace les mécanismes classiques de recherche aléatoire par des phénomènes quantiques. Le recuit quantique simulé s'appuie sur notamment les méthodes de Path Integral Monte-Carlo (PIMC) à la place de MCMC utilisé par le recuit simulé. Concrètement, le recuit quantique est basé sur théorème adiabatique en mécanique quantique. Avant de l'énoncer, commençons par définir ce qu'est un Hamiltonien.

Dans notre problème d'optimisation, un Hamiltonien représente une fonction de coût quantique, dont la valeur minimale est un état quantique d'énergie minimale ou état propre associé à la plus petite valeur propre de l'Hamiltonien, si on voit un Hamiltonien comme un opérateur et donc une matrice en algèbre linéaire. Donc l'Hamiltonien décrit notre système quantique, et le but est d'en trouver l'état fondamental, donc état de plus basse énergie qui correspond à la solution optimale de notre problème d'optimisation.

Le théorème adiabatique dit que :

"Un système physique reste dans son état propre instantané si une perturbation agissant sur lui évolue suffisamment lentement, et s'il existe un écart d'énergie (gap) entre la valeur propre correspondante et le reste du spectre de l'Hamiltonien."

Si on parle d'état propre minimal, alors on comprend que le système reste dans un état minimal lorsqu'on le fait évoluer. C'est à dire qu'en étant dans l'état qui minimise notre "fonction de coût" initiale, et qui serait connue, nous allons évoluer lentement vers un état qui minimise notre "fonction de coût" finale — la solution optimale.

L'astuce algorithmique consiste donc à formuler le problème d'optimisation comme la recherche de l'état fondamental d'un Hamiltonien $H_{problème}$ ou H_{final}, à démarrer dans l'état fondamental d'un Hamiltonien trivial H_0 ou $H_{initial}$, et à interpoler progressivement entre les deux. Le système évolue ainsi vers la solution optimale si l'évolution est suffisamment lente. L'Hamiltonien qu'on appellera Hamiltonien global de notre problème d'optimisation est donc formulé de la sorte : $H(t) = (1 - s(t)) \cdot H_{initial} + s(t) \cdot H_{final}$

Où $s(t)$ est une fonction croissante du temps allant de 0 à 1 qui est souvent simplement $s(t) = \frac{t}{T}$ pour un temps d'évolution T. Au début de l'évolution, $H(t) \approx H_{initial}$, dont l'état minimal est facile à préparer et à la fin, $H(t) \approx H_{final}$, dont l'état minimal est la solution au problème d'optimisation.

Pour utiliser efficacement le recuit quantique, il faut traduire notre problème d'optimisation dans une forme standard que la machine de recuit quantique peut comprendre et manipuler - la **formulation QUBO**, pour Quadratic Unconstrained Binary Optimization.

Un problème QUBO est défini par une fonction de coût quadratique sur des variables binaires, sans contrainte explicite : Minimiser $\sum_i Q_{ii} x_i + \sum_{i<j} Q_{ij} x_i x_j$ avec $x_i \in \{0,1\}$

Le tout peut être résumé dans une matrice symétrique Q, appelée matrice QUBO, où chaque élément Q_{ij} représente l'impact d'activer ensemble les variables x_i et x_j sur la fonction de coût. Le but est alors de trouver le vecteur binaire x qui minimise $x^T Qx$.

Cette formulation est très générale et permet de représenter une grande variété de problèmes combinatoires : sélection de sous-ensembles, couverture de sommets, planification, routage, etc.

D'un point de vue physique, chaque variable binaire x_i est encodée par un qubit, et les termes quadratiques $x_i x_j$ deviennent des interactions entre qubits dans l'Hamiltonien final H_{final}. En raison de cette relation étroite, les problèmes décrits par une fonction de coût (QUBO) ou un Hamiltonien de coût (Ising) peuvent être résolus en simulant le processus de recherche de leur énergie minimale.

La documentation de myQLM comporte des sections dédiées à la définition des Hamiltoniens de type Ising, des fonctions de coût QUBO ainsi que des problèmes combinatoires en général. Elle y explique comment les encoder, tout en précisant les conventions adoptées pour leur formulation exacte. Une fois encodés, ces problèmes peuvent être envoyés vers un **recuit simulé** pour exécution.

Exemple Max Cut

La définition du problème MaxCut est telle qu'on considère un graphe non orienté avec un ensemble de sommets V et un ensemble d'arêtes E. L'objectif est de partitionner le graphe en deux sous-graphes reliés par le plus grand nombre possible d'arêtes.

MaxCut est un problème de maximisation, et sa fonction de coût peut être reformulée en un problème de type Ising à l'aide de son Hamiltonien associé : $H = \sum_{uv \in E} s_u s_v$, où s_u est une variable de spin valant 1 si le sommet u appartient au premier sous-graphe et -1 s'il appartient au deuxième sous-graphe. Par un simple changement de variable, on peut retrouver nos variables binaires x_i de la formulation QUBO, à partir des spins s_i.

La plateforme myQLM permet d'encoder ce type de problème sous forme hamiltonienne à l'aide de la classe MaxCut, en fournissant simplement le graphe concerné. On peut ensuite générer un job à partir de ce problème et l'envoyer vers un solveur puissant, comme le recuit simulé, ou le recuit quantique simulé (SQA) si l'on utilise une Appliance Qaptiva.

Machines analogiques autres que le recuit quantique

Contrairement au recuit quantique, qui repose sur une interpolation adiabatique entre deux Hamiltoniens (souvent de type Ising), un QPU analogique hors recuit implémente directement un Hamiltonien cible, généralement conçu pour simuler un système physique donné, comme une molécule ou un matériau. Cet Hamiltonien peut inclure des termes de type Hubbard, Heisenberg, ou d'autres modèles d'interaction plus complexes. Un exemple de Hamiltonien Rydberg 2-niveaux :

$$H = \sum_i \left(\frac{\hbar\Omega}{2}\sigma_i^x - \hbar\Delta n_i\right) + \sum_{i<j} V_{ij} n_i n_j$$

Il ne s'agit pas ici de décrire en détail cet Hamiltonien, mais de rappeler que, dans un QPU analogique hors recuit quantique, l'évolution temporelle ne résulte pas nécessairement d'un Hamiltonien dépendant du temps (comme c'est le cas dans le recuit quantique), mais plutôt du fait que l'état quantique évolue naturellement sous l'effet d'un Hamiltonien constant (si l'on ne fait pas varier Ω, Δ ou les positions des

atomes). En général, H peut toujours être décomposé de la manière suivante : $H(t) = \sum_i \lambda_i(t) H_i$

Avec $\lambda_i(t)$ un champ de contrôle analogique dépendant du temps, et H_i un opérateur hermitien.

Les QPU analogiques, comme ceux basés sur des atomes de Rydberg, des ions piégés ou des systèmes photoniques, manipulent des systèmes dont les interactions physiques sont décrites par des **variables continues**, modélisées par des nombres réels. L'information quantique y est encodée directement dans ces interactions, sans passer par des portes logiques discrètes comme dans les QPU numériques.

Qaptiva fournit plusieurs émulateurs de QPU analogiques, avec ou sans bruit

Qaptiva est équipé de trois émulateurs de QPU analogiques qui partagent certaines fonctionnalités tout en présentant quelques différences clés.

AnalogQPU : Cet émulateur est basé sur Boost.odeint qui est une bibliothèque C++, principalement utilisée pour les ODE (équations aux dérivées ordinaires) dans divers domaines de la physique et de l'ingénierie. Cet émulateur est intrinsèquement rapide grâce à l'utilisation de solveurs C++/CUDA.

QutipQPU : Cet émulateur utilise la bibliothèque open-source QuTiP (Quantum Toolbox in Python) du langage Python. QuTiP est spécialisée dans la simulation des systèmes quantiques en prenant en charge la combinaison de différents types de Hamiltoniens :

- **Hamiltoniens bosoniques** : Les bosons sont des particules avec un spin entier et peuvent occuper le même état quantique. Les Hamiltoniens de bosons peuvent inclure des termes décrivant les interactions entre les bosons, comme dans les condensats de Bose-Einstein.
- **Hamiltoniens fermioniques** : Les fermions ont un spin demi-entier et obéissent au principe d'exclusion de Pauli, ce qui signifie qu'ils ne peuvent pas occuper le même état quantique. Les Hamiltoniens de fermions incluent des termes décrivant les interactions entre les fermions, souvent utilisés pour modéliser les électrons dans les matériaux.

QutipQPU est généralement plus lent qu'AnalogQPU. Cependant, au-delà des fonctionnalités d'AnalogQPU, on peut combiner les Hamiltoniens peut créer des modèles plus complexes qui décrivent des systèmes où les interactions entre les bosons et les fermions sont toutes prises en compte. Par exemple, dans certains matériaux, les interactions entre les électrons (fermions) et les phonons (bosons) peuvent être décrites par un Hamiltonien combiné.

MPSTraj : Cet émulateur est basé sur l'algorithme de trajectoire quantique et la représentation MPS des vecteurs d'état. La méthode des trajectoires quantiques est une méthode de Monte Carlo pour résoudre l'équation de Lindblad. Donc, ce n'est pas un émulateur exact. Pour augmenter encore plus l'efficacité, les états purs sont stockés sous forme d'états de produit matriciels (MPS), qui sont particulièrement efficaces pour compresser les états faiblement intriqués. Grâce à son utilisation de MPS, cet émulateur est capable d'atteindre un nombre de qubits beaucoup plus élevé que les émulateurs exacts tels que AnalogQPU. Pour une émulation en absence de bruit, MPSTraj calcule la dynamique unitaire donnée par l'Hamiltonien, ce qui en fait une version analogique de l'émulateur MPS.

Exemple de code avec QutipQPU

La simulation de systèmes à plusieurs corps (*many-body*) est extrêmement importante, tant en physique fondamentale que dans de nombreuses applications technologiques. Prenons un exemple très simple : deux électrons pouvant sauter d'un site à un autre. Le terme « site » désigne généralement une position discrète dans un réseau où une particule (typiquement un électron) peut se trouver. Supposons que l'Hamiltonien soit défini par :

$$H = c_0^\dagger c_1 + c_1^\dagger c_0$$

Où $c\dagger$ est un opérateur de création et c un opérateur d'annihilation.

Imaginons qu'au début, le système soit dans l'état $|01\rangle$, c'est-à-dire que le site 0 est vide et le site 1 est occupé par un électron. On laisse alors le système évoluer pendant une durée t, avant de mesurer l'occupation du site 0 à l'aide de l'opérateur $c_0^\dagger c_0$. En physique quantique, en particulier dans les systèmes de fermions, cet opérateur permet de mesurer l'occupation du site 0 dans un modèle de réseau. On s'attend à ce que l'occupation des deux sites ait évolué, de sorte que l'état du système soit devenu $|10\rangle$.

Voici comment représenter cet hamiltonien :

```
import numpy as np
from qat.qpus import QutipQPU
from qat.core import Observable, Term, Schedule
from qat.core.variables import Variable, sin

# Encode the Hamiltonian and create a Schedule
drive = Observable(2, pauli_terms=[Term(1, "Cc", [0, 1]), Term
(1, "Cc", [1, 0])])

# Supposons que l'Hamiltonien soit appliqué pendant une durée de π/2
unités de temps
schedule = Schedule(drive=drive, tmax=np.pi / 2)
```

Remarquons que le *schedule* (ou calendrier de contrôle) encode l'évolution d'un Hamiltonien sur une durée donnée. Nous allons mesurer $\langle c_0^\dagger c_0 \rangle$, la valeur d'espérance de l'observable $c_0^\dagger c_0$, à la fin de l'évolution. Cette observable est définie comme suit : Observable(2, pauli_terms=[Term(1, 'Cc', [0, 0])])), où la valeur 2 indique qu'il s'agit d'une observable de deux qubits.

```
# Construct a job with an observable to measure
job = schedule.to_job(psi_0="01", # start from the 0th site unoccupied
and the 1st - occupied
                      job_type="OBS", observable=Observable(2,
pauli_terms=[Term(1, 'Cc', [0, 0])]))
```

Remarquons que le « job » est composé d'un *schedule*, contrairement aux « jobs » qui sont composés de circuits quantiques dans le cas des QPU à portes.

Par défaut, si aucun modèle matériel (*Hardware Model*) n'est fourni en argument, le QPU effectue une émulation sans bruit.

```
# Create a QPU and submit the job
qpu = QutipQPU(nsteps=200) # can specify the number of time steps for
```

```
 the integration
res = qpu.submit(job)
print("<n_0 (pi/2)> =", res.value)

<n_0 (pi/2)> = 0.9999999999993716
```

$\langle c_0^\dagger c_0 \rangle = 1$, ce qui signifie que l'état du système est devenu $|10\rangle$; c'est exactement ce à quoi on s'attendait.

Exemple de code avec AnalogQPU

Prenons un exemple simple avec un seul qubit, d'indice 0, dont l'Hamiltonien qui varie avec le temps, est défini par :

$$H(t) = (1-t)\sigma_z^{(0)} + t\sigma_x^{(0)}$$

Où σ_0^x, σ_0^z sont respectivement les matrices de Pauli :

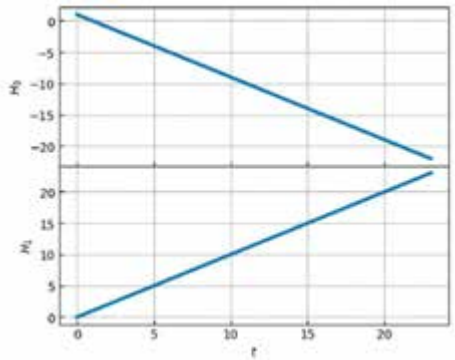
$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

Nous supposons que l'hamiltonien serait exécuté pour une durée de 23 unités de temps. Voici comment représenter ce H(t) :

```
%matplotlib inline
from qat.core import Variable, Schedule, Observable, Term

# Define a time Variable
t_variable = Variable("t")

# Encode the Hamiltonian and create a Schedule
hamiltonian = (1 - t_variable) * Observable(1, pauli_terms=[Term(1,
'Z', [0])]) + \
        t_variable * Observable(1, pauli_terms=[Term(1, 'X', [0])])
schedule = Schedule(drive=hamiltonian,
             tmax=23.0)
print(schedule)
schedule.display()
```

```
drive:
1 * (1 - t) * (Z|[0]) +
t * (X|[0])
```

Notons $H_0 = \lambda_0(t) = 1\text{-}t$, $H_1 = \lambda_1(t) = t$, nous avons les courbes :



Nous allons mesurer, $\langle \sigma_0^x \rangle$, la valeur d'espérance de l'observable σ_0^x à la fin de l'évolution. Cet observable est défini de la manière suivante :

```
H_target = Observable(1, pauli_terms=[Term(1, 'X', [0])])
job_obs = schedule.to_job(job_type="OBS", observable=H_target)
```

En absence de bruit, $\langle \sigma_0^x \rangle = \langle \Psi(t_f) \mid \sigma_0^x \mid \Psi(t_f) \rangle$, l'émulateur AnalogQPU done :

```
from qat.qpus import AnalogQPU

qpu = AnalogQPU() # no further arguments needed at this stage

# Measure the observable
res_obs = qpu.submit(job_obs)
print("<H_target> =", res_obs.value)
```

```
<H_target> = 0.1549594459807696
```

Soumission sur QPU

La solution Qaptiva d'Eviden est hardware agnostique permettant le développement d'algorithmes et d'exécution sur n'importe quel type de hardware en utilisant des outils (Plugins) pour intégrer les technologies spécifiques de tel ou tel QPU.

En pratique le QPU doit être encapsulé dans un objet myQLM, qui tient compte de sa topologie – nombre de qubits physiques, connectivité des qubits entre eux, ... L'utilité des plugins est aussi un avantage dans myQLM car au-delà d'instancier ce QPU, on peut l'étendre au travers de plugins qu'il est possible de développer – par exemple pour prendre en compte une contrainte de dernier qubit physique qui reçoit une opération dans ce même QPU.

Un exemple de module qui fait partie de la release la plus récente de myQLM à savoir la 1.11.3 (déjà existant dans notre Appliance Qaptiva) est le NISQCompiler [5] qui est sous forme de black-box regroupant certaines notions de plugins d'optimisation de circuit par exemple. Pour en tirer avantage, voici les étapes que ce module tout-en-un parcours avant d'envoyer un circuit compilé à un QPU : d'abord, les portes passées en entrée sont traduites en portes à 1 qubit et à 2 qubits. Ensuite, le circuit est optimisé pour réduire le nombre de portes d'intrication, tout en l'adaptant à la topologie de l'architecture. Enfin, les portes sont traduites dans l'ensemble de portes cible et compressées lorsque c'est possible.

myQLM est une suite logicielle gratuite de programmation quantique, accessible à tous. Elle est conçue pour l'apprentissage et offre des fonctionnalités avancées qui simplifient considérablement la programmation. Pour la télécharger et l'installer librement, il convient de se référer à la documentation en ligne [6]. Pour plus d'informations, consultez le lien : https://eviden.com/solutions/quantum-computing/

Références

[1] M. A. Nielsen and I. L. Chuang, *Quantum Computation and Quantum Information* (Cambridge University Press, Cambridge, 2010) ; A. Barenco *et al.*, *Elementary gates for quantum computation*, Phys. Rev. A **52**, 3457 (1995), https://arxiv.org/abs/quant-ph/9503016.

[2] Orus, R., A Practical Introduction to Tensor Networks: Matrix Product States and Projected Entangled Pair States. Annals of Physics 349, 117–158 (2014), https://arxiv.org/abs/1306.2164.

[3] E. Bernstein and U. Vazirani, *Quantum Complexity Theory*, SIAM J. Comput. **26**, 1411 (1997).

[4] "Ising formulations of many NP problems", A. Lucas, Frontiers in Physics, 2014. DOI: 10.3389/fphy.2014.00005

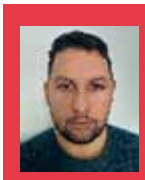
[5] https://myqlm.github.io/04_api_reference/module_qat/module_plugins/nisqcompiler.html#qat.plugins.NISQCompiler

[6] https://myqlm.github.io/01_getting_started/:myqlm:01_install.html

Eviden Quantum Computing Consulting:
Robert Wang, Damien Nicolazic, Mariem Bahri, Grégory Vieux, Olivier Hess

Sécuriser mon réseau informatique avec l'informatique quantique : comparaison des solutions Dwave, Qiskit et Pasqal

EURO QUANTUM Operations Research



Ali Abbassi
Actuellement en thèse à l'UTT en collaboration avec la société Orange dans le cadre d'un contrat ANRT. Email : ali.abbassi@utt.fr

Yann Dujardin
Chercheur en computer sceince et optimisation chez Orange Innovation. Email : yann.dujardin@orange.com

Philippe Lacomme
Professeur à l'ISIMA où il enseigne l'optimisation et l'informatique quantique. Email : philippe.lacomme@isima.fr

Caroline Prodhon
Professeur à l'UTT. Email : caroline.prodhon@utt.fr

1 INTRODUCTION

La sécurisation d'un réseau : c'est quoi exactement ?

On entend souvent parler de cybersécurité… mais que recouvre ce terme ?

Il s'agit d'éléments variés tels que les mesures techniques et juridiques pour protéger des réseaux, des données et les utilisateurs. Ici on s'intéresse plus spécifiquement à une question simple : comment sécuriser un réseau en cas d'intrusion.

Que signifie sécuriser un réseau informatique en cas d'intrusion ?

C'est isoler le plus rapidement possible les ordinateurs sensibles, en coupant leur accès.

Prenons de l'exemple de la **figure 1**, si l'on souhaite protéger le serveur LDAP (Lightweight Directory Access Protocol) très sensible de l'entreprise, alors que le routeur 1 a été compromis par un pirate informatique, il est possible de couper ainsi un chemin d'accès du routeur 1 vers le serveur LDAP en déconnectant le routeur 4.

Comme il n'existe plus aucun chemin pour joindre le serveur LDAP, la déconnexion du routeur 4 est une action adaptée pour préserver l'intégrité des données du serveur LDAP.

La protection d'un réseau peut ainsi sembler simple, mais dans la réalité, couper un routeur signifie faire une contre-mesure qui va stopper les flux d'information mais qui coute cher en temps, en argent… et nuit aux usagers. Il faut donc limiter les déconnexions. Or, ceci n'est pas trivial, car dans le cas général il faut considérer :

- Des centaines ou des milliers de routeurs dans le système d'informations ;
- Des dizaines de serveurs sensibles dont on doit garantir l'intégrité ;
- Des milliers de chemins possibles entre le pirate d'un côté qui a corrompu un nœud de l'organisation et les serveurs sensibles à protéger.

Face à ce défi, il est essentiel de trouver des solutions intelligentes et efficaces pour sécuriser les réseaux sans compromettre leur fonctionnement. C'est là que les technologies quantiques, comme celles proposées par Dwave, Qiskit et Pasqal, peuvent offrir des alternatives innovantes.

Que cherche-t-on à faire ?

A SECURISER AU MIEUX, c'est-à-dire en minimisant l'impact sur l'entreprise de la déconnexion des routeurs.

Formalisons l'idée

Comme il est impossible de déconnecter tous les routeurs dans un temps raisonnable et sans engendrer des coûts prohibitifs ni de problème vis-à-vis des clients, il nous faut un peu formaliser le problème en nous posant les bonnes questions :

- Définir le réseau à sécuriser ;
- Formuler mathématiquement le problème de sa sécurisation ;
- Proposer une résolution quantique du problème à l'aide de trois types d'ordinateurs quantiques.

Considérons une version simplifiée d'un réseau : un **graphe**, c'est-à-dire un ensemble de **nœuds** (ou sommets), reliés ou non par des **arêtes** (ou liens). Cette représentation abstraite nous dispense de détails techniques : les nœuds peuvent représenter des ordinateurs, des routeurs, des consoles, ou tout autre équipement électronique ; les arêtes, quant à elles, peuvent modéliser des connexions physiques comme des

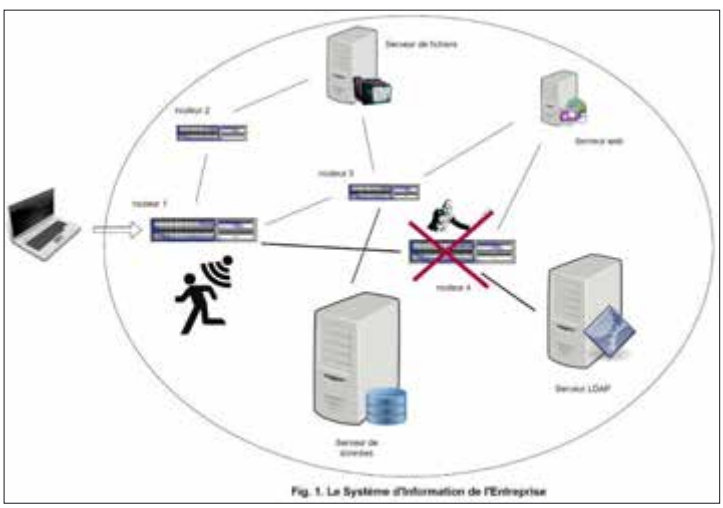


Fig. 1. Le Système d'Information de l'Entreprise

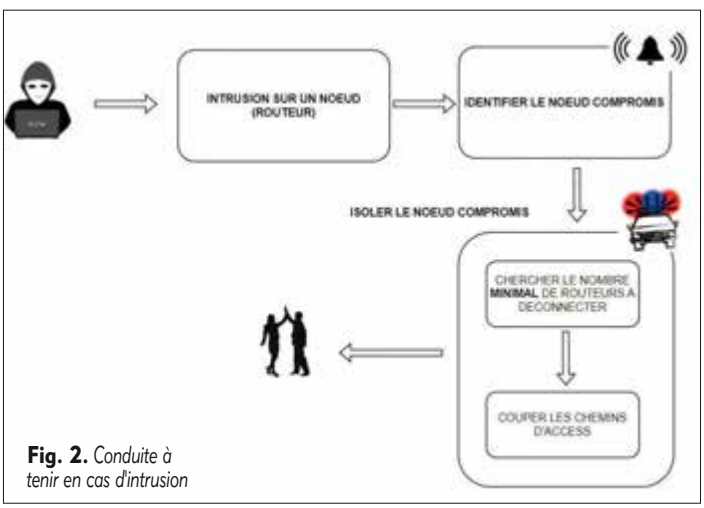


Fig. 2. *Conduite à tenir en cas d'intrusion*

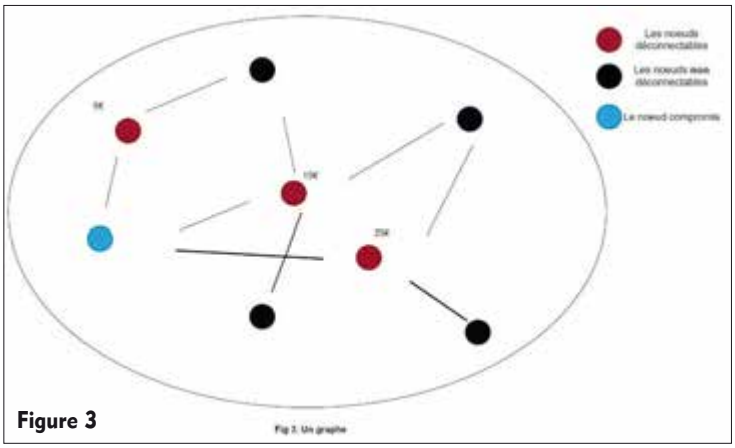


Figure 3

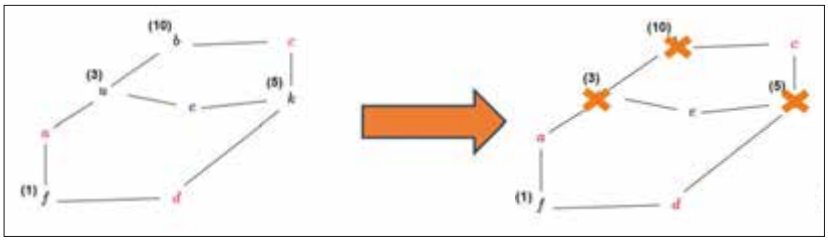


Figure 4 : *exemple de sécurisation d'un réseau représenté par un graphe.*

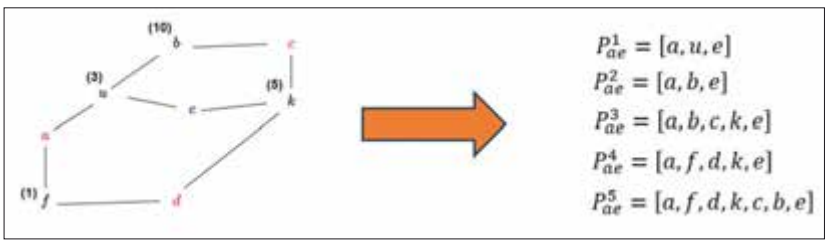


Figure 5 : *énumération de tous les chemins élémentaires entre les nœuds a et e.*

Fig. 6. *Trois technologies D-Wave (photo © D-Wave Systems), Pasqal (photo © Pasqal), IBM Q (photo © IBM).*

câbles ou des connexions wifi. Ce qui nous importe, c'est la structure topologique du réseau : **qui est connecté à qui ?** Cela suffit à poser notre problème. **Figure 3**

Le scénario d'attaque

Imaginons maintenant qu'un cyberattaquant parvienne à prendre le contrôle de certains nœuds du réseau — par exemple en compromettant un ordinateur — et tente ensuite de se propager pour atteindre des zones critiques. Ces zones peuvent contenir des données sensibles ou contrôler des fonctions importantes du système.

Notre objectif est alors le suivant : **empêcher l'attaquant d'atteindre ces zones critiques**, en installant des **contre-mesures** sur certains nœuds intermédiaires (par exemple, des firewalls ou systèmes de détection), afin de bloquer les chemins d'accès.

Une première modélisation

Dans notre modèle, nous désignons certains nœuds comme **sources** (points d'entrée possibles pour l'attaque), et d'autres comme **cibles** (points qu'il faut impérativement protéger). La stratégie consiste à **interrompre tous les chemins** qui relient des sources à des cibles, en supprimant des nœuds intermédiaires par la mise en place de contre-mesures. **Figure 4**

Prenons un petit graphe dans lequel les nœuds a, c et d (en rouge) sont les points d'entrée de l'attaquant, et e (en bleu) est un nœud critique à protéger. Certains nœuds comme u, b, k et f portent un coût associé à la pose d'une contre-mesure (noté par un nombre). Cela reflète l'idée que protéger un nœud a un coût (énergétique, financier, technique, etc.)

Les terminaux (sources et cibles) **ne peuvent pas être supprimés**, car ils sont nécessaires au fonctionnement du réseau. C'est pourquoi ils n'ont pas de coût.

Objectif : couper les chemins au moindre cout

Pour que le réseau soit protégé, **il faut couper tous les chemins** entre nœuds sources et nœuds cibles. Dans notre exemple, cela signifie placer des contre-mesures sur u, b et k de manière à rendre inaccessible le nœud e depuis a, c ou d. Problème résolu ? Presque… car une difficulté majeure se pose.

Une complexité qui explose !

À mesure que le nombre de nœuds, de sources et de cibles augmente, le nombre de chemins à couper explose. Même dans notre petit réseau de 8 nœuds, il existe déjà 5 chemins élémentaires reliant rien que la source a à la cible e (des chemins sans répétition de nœuds). Cela peut sembler peu, mais imaginez un réseau de taille réelle, comme celui d'un opérateur télécom : des milliers de nœuds et des millions de chemins possibles ! **Figure 5**

Ce problème devient alors difficile à résoudre par des méthodes classiques. C'est un problème combinatoire dont une variante où l'on associe à chaque source une unique cible s'appelle le Vertex Minimum Multicut.

- "Vertex", car on agit sur les nœuds (et non sur les arêtes) ;
- "Minimum", car on cherche à bloquer un nombre minimal de nœuds ;
- "Multicut", car il s'agit de séparer plusieurs paires source-cible.

Face à cette explosion combinatoire, on se demande si l'ordinateur quantique peut nous offrir un avantage. Grâce à des phénomènes comme la **superposition**, l'**interférence** ou l'**intrication**, un processeur quantique pourrait — en théorie — **explorer simultanément plusieurs chemins** dans le graphe et décider des coupes optimales plus efficacement. **Fig. 6.**

C'est ce que nous allons tester avec trois types d'ordinateurs quantiques : Le **quantum annealer de D-Wave**, l'ordinateur de **PASQAL** et celui **d'IBM**

2 UN PEU DE MATHÉMATIQUES **Figure 7**

Faisons simple.. enfin presque ! Nous allons modéliser notre problème sous forme d'un programme mathématique.

Figure 10 : *le code de Dware à partager sans modération*

Soit $G = (V, E)$ un graphe générique, avec V un ensemble de nœuds (Vertex) et E un ensemble d'arêtes (Edges). Aussi, notons :

v_1 : **un sommet source** (le nœud 1)

v_2 : **un sommet cible** (le nœud 3)

$v_3 : V - \{v_1, v_2\}$: **les sommets non cibles et non source**

$S_i : i \in V_3 \rightarrow R^+$, le cout associé

$P^k_{v_1,v_2}$ chemin élémentaire numéro k qui va du sommet v_1 vers le sommet v_2

$n^k_{v_1,v_2}$: nombre de sommet du chemin k

$y^{k,u}_{v_1,v_2} = 1$ si $u \in P^k_{v_1,v_2}$ et 0 sinon

Les variables du problème sont : $x_i = 0$ si le sommet i est présent, $i \in V_3$

Nous pouvons écrire la fonction-objectif : $\min \sum_{i=1}^{|V_3|} x_i s_i$

qui doit respecter l'ensemble de contraintes :

$$\forall k = 1, \ldots, n^k_{v_1,v_2} \qquad \sum_{u \in V_3} y^{ku}_{v_1,v_2} . x_u \geq 1$$

Si on détaille le modèle générique sur le cas particulier de la **figure 6**, voici ce qu'on obtient.

$S_2 = 10$ c'est-à-dire que fermer le nœud 2 coûte 10€, $S_4 = 20$ c'est-à-dire que fermer le nœud 4 coûte 20€

$S_5 = 40$ c'est-à-dire que fermer le nœud 5 coûte 40€, $S_6 = 30$ c'est-à-dire que fermer le nœud 6 coûte 30€

Pour aller du nœud 1 compromis par le pirate au nœud 3, nous avons 2 chemins possibles :

Chemin 1. Nœud 1 – Nœud 2 – Nœud 4 – Nœud 3 ;

Chemin 2. Nœud 1 – Nœud 5 – Nœud 6 – Nœud 3

Mathématiquement ceci se note : $P^1_{1,3} = \{2,3\}$

et $P^2_{1,3} = \{5,6\}$

ainsi que $y^{1,2}_{1,3} = 1$ et $y^{2,4}_{1,3} = 1$

La fonction objectif vaut :

$$\min_{x \in \{0,1\}^5} f(x) = min(10x_2 + 20x_4 + 40x_5 + 30x_6)$$

et les contraintes sont $x_2 + x_4 \geq 1$ et $x_5 + x_6 \geq 1$

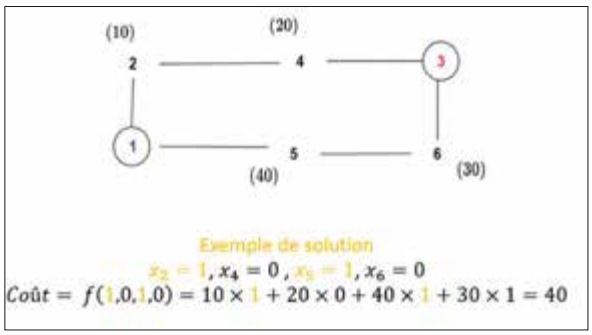


Figure 7 :
Un problème et une solution

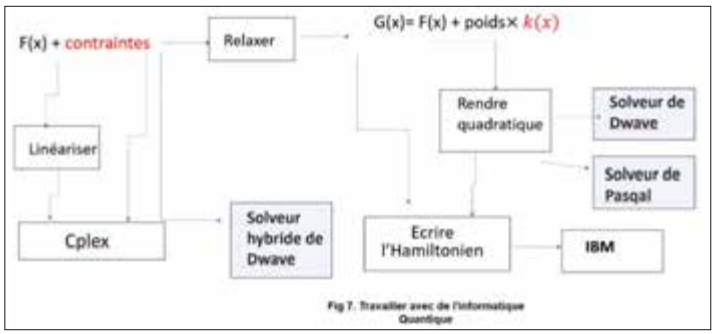


Figure 8

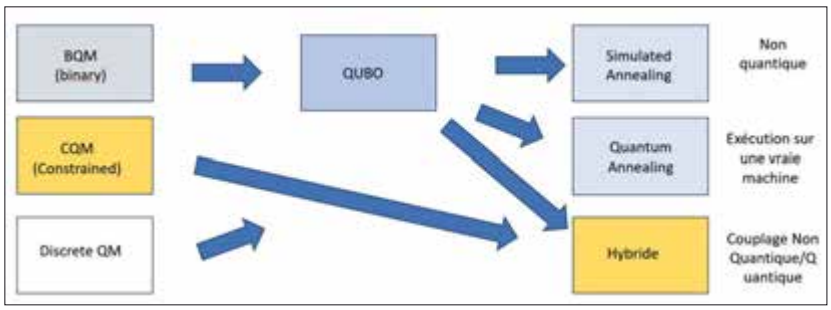


Figure 9
Choisissez vos librairies

3 UN PEU D'INFORMATIQUE

Les outils

La forme mathématique que l'on a vient de voir est linéaire. Or certains ordinateurs quantiques n'acceptent que des formes quadratiques comme Pasqal ou Dwave, d'autres acceptent des formes qui sont plus que quadratiques. Il faut donc s'adapter aux outils et exprimer le problème sous une forme particulière. On va voir quelles sont les difficultés inhérentes à ces contraintes techniques. **Figure 8**

Utiliser Dwave : des difficulté de modélisation mais surtout d'accès.

Dwave exécute la méthode Quantum Annealing à partir d'une librairie appelée BQM. **Figure 9**

Mais que fait BQM ? Il vous permet de modéliser votre modèle mathématique ! **Figure 10**

Et là c'est le graal… on exécute son code sur une vraie machine !

Mais derrière cette apparente facilité, se cache un frein non négligeable : La machine n'accepte que certaines formulations mathématiques dites 'QUBO ' avec des restrictions sur le réseau considéré en plus d'une Impossiblité depuis quelques mois d'avoir gratuitement du temps de calcul sur Dwave. La politique de la firme américaine est devenue très restrictive. Si vous disposez d'un peu de ressource, on peut acheter du temps de calcul et faire quelques tests. Attention à bien évaluer vos besoins !

Utiliser Qiskit : bonne nouvelle mais….

Qiskit proposé par IBM propose un accès gratuit à des simulateur des vraies machines mais hélas on ne peut pas comme avec Dwave donner "directement" son modèle. Non, il faut le réécrire pour obtenir un Hamiltonien avec des portes… quantiques bien sûr.

Les portes s'écrivent comme des matrices (des tableaux de nombres en quelques sortes), Par exemple (c'est quoi cet exemple ? peut-on décrire un peu ce qu'on fait ?) :

$$\frac{1}{2}(Id - Z) = \frac{1}{2}\left(\begin{pmatrix}1 & 0\\ 0 & 1\end{pmatrix} - \begin{pmatrix}1 & 0\\ 0 & -1\end{pmatrix}\right) = \frac{1}{2}\begin{pmatrix}1-1 & 0-0\\ 0-0 & 1-(-1)\end{pmatrix} = \frac{1}{2}\begin{pmatrix}0 & 0\\ 0 & 2\end{pmatrix} = \begin{pmatrix}0 & 0\\ 0 & 1\end{pmatrix}$$

Et de même pour l'addition cette fois

$$\frac{1}{2}(Id + Z) = \begin{pmatrix}1 & 0\\ 0 & 0\end{pmatrix}$$

Un Hamiltonien c'est une formule composée de portes Z par exemple.

Que faut-il faire pour transformer notre modèle mathématique en Hamiltonien ?

Il faut d'abord prendre notre fonction objective et remplacer les x_i par des $\frac{1}{2}(Id + Z)$ sans oublier les contraintes. Et pour continuer, il faut coder une approximation du Quantum Annealing qui est fourni nativement sur la machine analogique de Dwave. Vous pouvez vous amuser à le faire. Voici le code d'une fonction objectif traduite en hamiltonien

$$(1-x_1).(1-x_2) = 1 <=> 1 - x_1 - x_2 + x_1x_2 = 0 => Id - \frac{1}{4}(Id + Z_1)(Id + Z_2) = Hamiltonien$$

Approximer cela veut dire représenter $H = \alpha H_D + (1-\alpha)H_P$ **qui va nous permettre de passer lentement de H_D un Hamiltonien dans son état d'Energie le plus bas vers H_P l'Hamiltonien de notre problème dans son état d'energie le plus bas aussi qui se traduit ensuite en la solution de notre problème. (Le minimum !) Et enfin il nous faut calculer $e^{-i.t.H}$ Fig. 11**

Est-ce que cela fonctionne ? mais oui bien sûr. Sur des exemples simples comme ceux montrés dans cet article on peut trouver 80% de la distribution de probabilités sur la solution optimale.

Utiliser Pasqal : ça ne marche pas pour tous les problèmes….

On prend ici un autre exemple de réseau, avec un nœud A comme source et un nœud F comme cible. On écrit notre problème sous forme de QUBO (Quadratic Unconstrained Binary Optimization), une manière standard de formuler des problèmes d'optimisation. Ce QUBO peut lui-même être représenté sous forme de graphe. Et une question importante se pose : **peut-on représenter ce graphe comme un ensemble d'atomes disposés sur un plan ? Figures 12 et 13**

Pourquoi cette question ? Parce que pour utiliser l'ordinateur quantique développé par **Pasqal**, chaque nœud de notre problème est représenté par un **atome piégé**. Si deux atomes sont placés à une certaine distance (ni trop proches, ni trop éloignés), alors on peut dire qu'ils sont reliés — au sens d'une arête dans un graphe.

Vous voyez où est la difficulté ? Est-ce qu'on peut, peu importe le QUBO dont on dispose, **toujours** le représenter sous la forme d'atomes physiquement disposés sur un plan ? La réponse est **non**, et c'est ce que montre l'exemple donné dans la **figure 14**. Certaines configurations de graphe sont tout simplement impossibles à réaliser avec cette technologie.

Dans le schéma encadré en noir, on voit ce qui correspond au processus adiabatique utilisé sur l'ordinateur quantique de Pasqal. Cela consiste à faire varier finement des lasers pour que le système évolue petit à petit vers son **état d'énergie**

```
Measurement Results:
{'00': 801, '01': 71, '11': 85, '10': 67}
Press any key to continue . . .
```

Figure 12 : *résultats obtenus sur Qiskit*

Définir le circuit

```
import qiskit

from qiskit import ClassicalRegister, QuantumCircuit
from qiskit import QuantumRegister
from qiskit import *

qreg_q = QuantumRegister(2, 'q')
qreg_c = ClassicalRegister(2,'c')

qc = QuantumCircuit(qreg_q,qreg_c)
```

Définir le HD

```
qc.h(qreg_q[0])
qc.h(qreg_q[1])
```

Quantum Simulated Annealing

```
t = 1

delta_t = 0.1
iter=0
```

Initialisation du paramètre t

```
while (t >= delta_t):
    iter = iter +1
    if (iter%100 == 0):
        print (str(iter)," ",str(t))

    # HD avec ZX
    qc.rx(-2*t, qreg_q[0])
    qc.rx(-2*t, qreg_q[1])

    # HP

    qc.rz(-2*1/4*(1-t), qreg_q[0])

    qc.rz(-2*1/4*(1-t), qreg_q[1])

    # de 0 vers 1
    qc.cx(qreg_q[0], qreg_q[1])
    qc.rz(-2*1/4*(1-t), qreg_q[1])
    qc.cx(qreg_q[0], qreg_q[1])
    t = t - delta_t
```

On implémente la porte quantique suivante

$$e^{-i(-X)t} = R_X(-2t)$$

$$H_P = Id - (Id + Z_0 + Z_1 + Z_0Z_1)$$

$$U_P = e^{-iH_P(1-t)} = e^{-iH_P(1-t)} = e^{-it(-\frac{1}{4}Z_0)}e^{-it(-\frac{1}{4}Z_1)}e^{-it(-\frac{1}{4}Z_0Z_1)}$$

$$U_P = R_Z(-2\tfrac{1}{4}t)CX_{01}.\left(Id \otimes R_Z\left(-2\tfrac{1}{4}t\right)\right).CX_{01}R_Z(-2\tfrac{1}{4}t)$$

Fig. 11. *Codez votre Quantum Annealing avec Qiskit d'IBM*

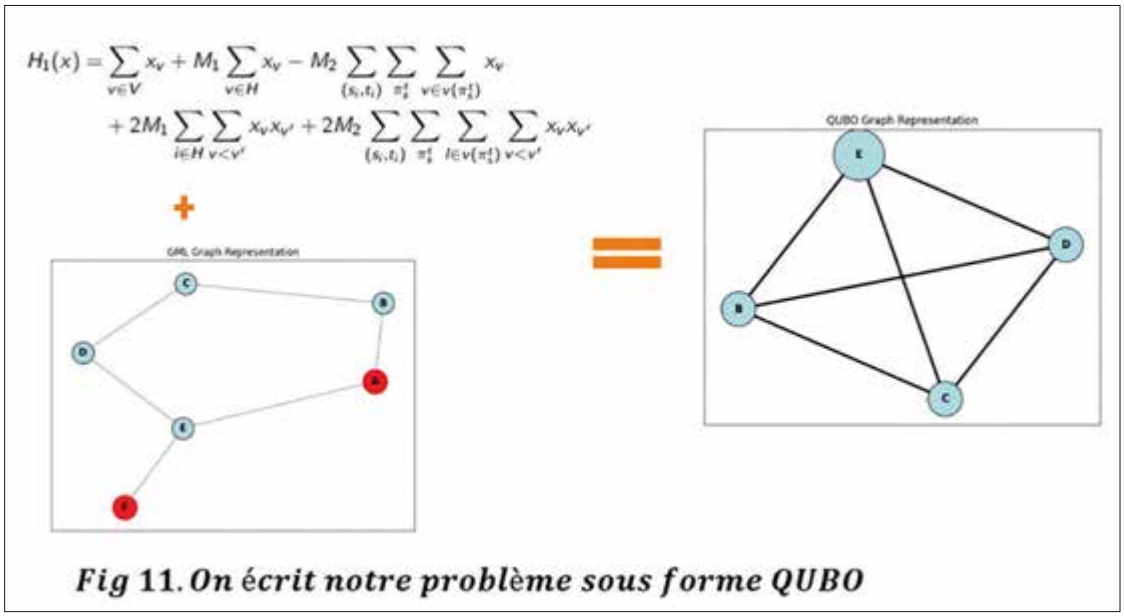


Fig 11. On écrit notre problème sous forme QUBO

Figures 13 : *on écrit notre problème sous forme QUBO*

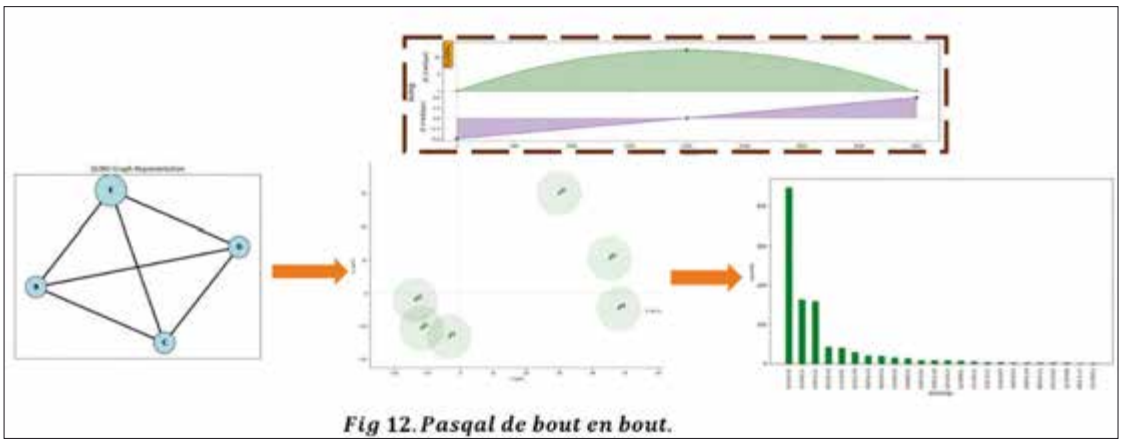


Fig 12. Pasqal de bout en bout.

Figures 14 : *Pasqal de bout en bout*

minimale, qui correspond à la solution du problème. Ce processus repose sur une transformation : on passe du QUBO à une configuration d'atomes sur un plan, tout en respectant deux contraintes physiques essentielles.
Première contrainte : les atomes ne doivent pas être trop proches les uns des autres. Sinon, ils interagissent de manière incontrôlée.

Deuxième contrainte : tous les atomes doivent être placés **dans un rayon de 50 micromètres** autour d'un point central. Au-delà, le système ne fonctionne plus correctement.
Dans un monde parfait (sans bruit, sans erreur), on prépare cette configuration d'atomes, on laisse le système évoluer, puis on le **"lit" plusieurs fois**. À chaque lecture, on obtient une solution possible, et on espère que **la bonne solution** est celle qui apparaît le plus souvent. Ce n'est pas toujours le cas, mais ça fonctionne bien dans certains contextes.
Par exemple, le problème appelé **Maximal Independent Set sur graphe de type "unit disk"** (même si le nom fait un peu peur !) fonctionne généralement **mieux** que l'exemple qu'on utilise ici. Cela montre que tous les problèmes ne sont pas également adaptés à ce type de machine.

5 CONCLUSION

Cette petite introduction devrait vous donner envie d'en savoir plus. Voici une liste de publications réalisées par Ali Abbassi et qui sont consultables en ligne.
Ali Abbassi, Yann Dujardin, Eric Gourdin, Philippe Lacomme, Caroline Prodhon : Assessing Quantum Annealing to solve the Minimum vertex multicut, CoDIT 2025
Ali Abbassi, Yann Dujardin, Eric Gourdin, Philippe Lacomme, Caroline Prodhon : Quantum Annealing To SolveNP-Hard Cut Problems, CIGI QUALITA MOSIM 2025

6 POUR EN SAVOIR PLUS

Ali est le webmaster d'un groupe de travail sur l'informatique quantique. Très orienté recherche, il contient aussi un forum où on peut trouver des offres de stages.
https://quantum-or.euro-online.org/

Simon Fried

VP en charge du développement commercial et marketing chez Classiq Technologies

Cofondateur de projets et produits deep tech, Simon s'attache depuis plusieurs années à concevoir des outils pensés pour les développeurs, qu'ils soient matériels ou logiciels, et à traduire les technologies émergentes en solutions concrètes pour les ingénieurs.

Combler le fossé entre langages classiques et programmation quantique de haut niveau

Revenons quelques décennies en arrière, à l'époque où la programmation informatique n'était encore qu'à ses prémices. À cette époque, l'informatique classique reposait sur de laborieuses opérations de câblage manuel de transistors. Pourtant cette époque était aussi marquée par une effervescence intellectuelle, où les nouvelles idées fleurissaient et où la programmation devenait progressivement plus efficace, plus simple et plus accessible.

Cela a finalement conduit à l'émergence des langages de haut niveau (HLL), qui ont libéré les développeurs des contraintes matérielles. Des langages comme C, Python ou Java permettent désormais de penser en termes de fonctions, de types et de modules, tandis que les compilateurs se chargent des registres et des codes opérationnels. Bien qu'il s'agisse d'un domaine nouveau, historiquement réservée à un cercle restreint de spécialistes, la programmation quantique entame la même transition. De nouvelles approches voient le jour au sein de la pile logicielle quantique, visant à rendre le développement quantique plus productif et plus accessible. C'est dans cette logique que s'inscrit le Qmod, le langage quantique de plus haut niveau. Développé par Classiq, il permet aux développeurs de dépasser la manipulation directe de portes et de la comptabilité des qubits, en fournissant une interface de type HLL adaptée à la logique quantique. Le développement d'ordinateurs quantiques devient ainsi une compétence plus accessible que jamais, et n'est plus réservé aux physiciens quantiques.

Comment les langages HLL ont-t-il changé la donne pour les développeurs ?

- **Abstraction** : Un même code, une fois exprimé, pouvait être exécuté sur n'importe quelle architecture CPU.
- **Productivité** : La gestion de la mémoire, l'ordonnancement des instructions et l'optimisation étaient délégués au compilateur.
- **Transfert de compétences** : La syntaxe et les bibliothèques partagées ont favorisé la création de communautés et de compétences réutilisables, d'un projet ou d'une plateforme à l'autre

Tout comme les langages HLL ont révolutionné la programmation classique, Qmod transforme aujourd'hui le développement pour les ordinateurs quantiques

- **Description fonctionnelle** : il reste encore quelques nouveaux concepts à comprendre et à apprendre, mais au lieu de penser en séquences de portes CX et RZ, les développeurs peuvent exprimer directement leur intention mathématique : « additionner deux registres », « appliquer la QFT », « optimiser sous ces contraintes ».
- **Indépendance matérielle** : le compilateur de Classiq mappe un seul modèle Qmod vers différentes modalités de back-ends: supraconducteurs, ions piégés, atomes neutres ou qubits cat, à l'image des compilateurs classiques qui ciblent x86 ou ARM.
- **Gestion automatique des ressources** : le moteur de synthèse prend en charge les qubits temporaires, le recyclage des ancilles et bien d'autres aspects, comme le font les compilateurs classiques pour la gestion des piles et des registres.
- **L'utilisateur définit les contraintes** : besoin d'un circuit quantique qui s'adapte à 100 qubits ? Il suffit de le déclarer comme contrainte et le compilateur explorera les variantes afin de produire un circuit optimal.

Certes, de nouveaux types de données doivent être pris en compte, mais les concepts restent familiers.

Comme Python, Qmod privilégie la lisibilité et la familiarité, à travers des mots-clés :

Classicaques HLL	Équivalent Qmod
Fonction	Fonction quantique (@qfunc) retournant des variables quantiques
Types de variables (int, float)	Nombres quantiques (QNum), tableaux de qubits (QArray)
For / while loops	repeat() : itérateur sur indices en surposition
If else	control(): condition basée sur la valeur d'un qubit

Voici un exemple concret d'addition modulaire :
Si, en Python un développeur écrit : z = (x + y) % N. L'équivalent en Qmod serait :

```
@qfunc
def mod_add(x: QNum, y: QNum, N: int, z: Output[QNum]):
    z = (x + y) % N
```

Cette seule ligne de HLL dissimule des dizaines de portes quantiques. Grâce au compilateur, le choix entre deux types d'additionneurs, l'un à propagation de retenue, l'autre basé

sur la QFT (Quantum Fourier Transform) se fait automatiquement, en fonction des ressources disponibles, comme le nombre de qubits. Sans cette abstraction, un développeur devrait écrire tout le circuit à la main, en bas niveau. Et même dans le cas d'un circuit relativement simple, il serait très difficile de savoir quel type d'additionneur est le mieux adapté à la machine cible.

Voici une illustration des étapes : **figures 1, 2, 3**

Bien que les HLL fassent progressivement leur apparition en programmation quantique, les développeurs ont encore beaucoup à assimiler pour devenir experts dans ce domaine, notamment :

- **Les notions fondamentales de l'information quantique** : superposition, intrication et effondrement lors de la mesure.
- **Les bases de l'algèbre linéaire** : les états quantiques s'expriment sous forme de vecteurs, les portes quantiques sous forme de matrices. Une bonne maîtrise des produits tensoriels facilitera grandement le débogage.
- **La compréhension des modèles de bruit et d'erreur** : contrairement aux processeurs classiques déterministes, les qubits actuels fonctionnent de manière probabiliste. Il est crucial de bien saisir l'importance du nombre de tirs (exécutions du circuit) et l'impact des taux d'erreur sur la précision des résultats, afin de paramétrer au mieux les contraintes dans Qmod.
- **La prise en compte des coûts opérationnels** : le nombre de portes à deux qubits, la profondeur du circuit et le nombre total de qubits influent directement sur la complexité temporelle et spatiale des calculs.

Premiers pas

Pour commencer à explorer cette nouvelle voie, nous vous recommandons de vous familiariser avec les bases de la physique quantique, de rafraîchir vos connaissances en algèbre linéaire, puis de parcourir les ressources disponibles dans la bibliothèque GitHub de Classiq où vous trouverez des modèles et exemples, comme la célèbre recherche de Grover pour les données non structurées, ou encore de nouvelles approches pour l'optimisation de portefeuille. Si vous souhaitez explorer la programmation bas niveau, vous pouvez également consulter les bibliothèques Qiskit. Pas besoin de vous précipiter pour lancer des tâches sur du matériel quantique en cloud : vous pouvez rapidement faire vos premiers tests sur un simulateur local.

Conclusion

Les langages de haut niveau n'ont pas seulement facilité le développement classique, ils ont également ouvert la voie à des logiciels d'une ampleur sans précédent, tout en ouvrant la porte à des experts de divers domaines. Les HLL quantiques, telles que Qmod, jouent un rôle similaire : ils gardent le modèle fonctionnel proche de la programmation quotidienne tout en masquant la complexité quantique sous-jacente. Pour un développeur maîtrisant Python ou C++, la transition vers la programmation quantique peut désormais se faire en quelques semaines, au lieu de plusieurs années. Devenir un véritable expert prendra évidemment plus de temps, mais ces langages ouvrent grand les portes du codage quantique et offrent de nouvelles opportunités de carrière.

Figure 1 : *Exemple d'arithmétique quantique de haut niveau dans Qmod de Classiq : cette fonction qfunc définit des variables quantiques, attribue des valeurs classiques et calcule un résultat quantique à l'aide d'opérations arithmétiques, démontrant ainsi comment une logique complexe peut être synthétisée en circuits quantiques à partir d'un code simple et lisible.*

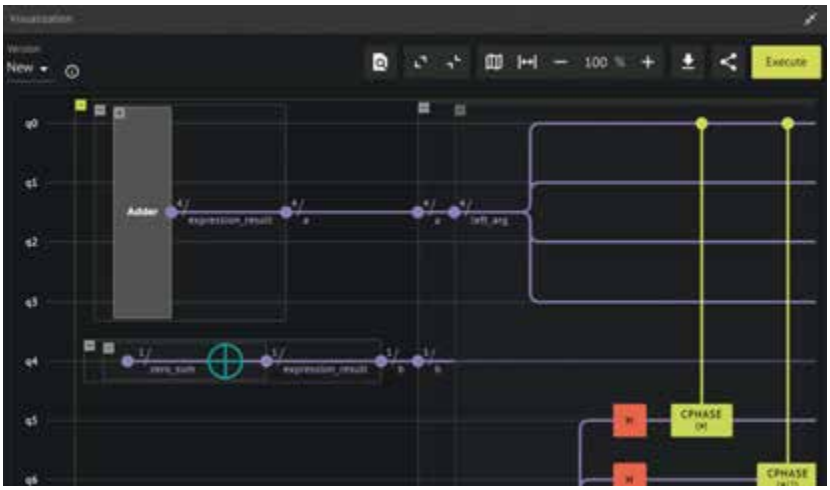


Figure 2 : *Visualisation d'un circuit quantique dans Classiq : représentation graphique d'un circuit synthétisé, illustrant des opérations logiques comme l'addition, les phases de contrôle et l'intrication à travers plusieurs qubits.*

Figure 3 : *un histogramme montre la distribution des états quantiques mesurés à partir d'un circuit quantique simulé, révélant les probabilités associées à différentes configurations de qubits.*

QAOA : une nouvelle manière de chercher des bonnes solutions et de les trouver

Philippe Lacomme

Professeur à l'ISIMA où il enseigne l'optimisation et l'informatique quantique. Email : philippe.lacomme@isima.fr

Éric Bourreau

Maître de Conférences au LIRMM de Montpellier et il coresponsable du groupe de travail sur l'informatique quantique au sein du GDR RO. Email : Eric.Bourreau@lirmm.fr

Gérard Fleury

Maître de Conférences associé au LIMOS. Il est spécialiste de statistique et d'optimisation. Il est coauteur de deux livres sur le quantique parus en 2022 et 2023. Email : gerard.fleury@isima.fr

Martin Bombardelli

Actuellement en thèse au LIMOS en collaboration avec la société CGI dans le cadre d'un contrat ANRT. Email : martin.bombardelli@cgi.com

Marc Sevaux

Professeur à Lorient au Lab-STICC et il coresponsable du groupe de travail sur l'informatique quantique au sein du GDR RO. Email : marc.sevaux@univ-ubs.fr

1 - La combinatoire nous rattrape

Où est le problème ?

Nous avons déjà présenté ce problème dans un numéro précédent. Tentons de résumer en quelques lignes les points importants. L'intérêt de l'informatique quantique est de résoudre dans des temps de calcul beaucoup plus courts des problèmes qui nécessitent actuellement des heures de temps de calcul.

Imaginons un commercial qui doit visiter un certain nombre de clients et revenir à son point de départ, en parcourant la plus courte distance possible. Une solution est une suite de villes qui commence à 0 (le dépôt) et qui se termine au dépôt (le nœud numéro 0).

```
Une_Solution = [0]*(w+1)
Une_Solution=[0, 3, 2, 1, 4, 7 ,5, 6, 0]
```

```
def cout_solution (liste, distance):
    cout = 0
    for i in range (0, w-1):
      cout = cout + distance[liste[i]][liste[i+1]]
    return cout

Cout = cout_solution (Une_Solution, distance)
print(Cout)
```

Quel est son coût ? Simplement la somme des distances.

Combien de solutions sont possibles ?

À mesure que le nombre de villes augmente, le nombre de solutions possibles explose puisque

Fig 1. Évolution du nombre de solutions, pour n villes, le nombre de circuits est de l'ordre de $n!$. Si l'on reprend le problème précédent à 7 clients, la première case du tableau **Solution** contient obligatoirement 0. La deuxième case du tableau peut contenir n'importe quel élément parmi les 7 restants, la troisième case n'importe quel élément parmi les 6 restants et ainsi de suite (voir **figure 1**).

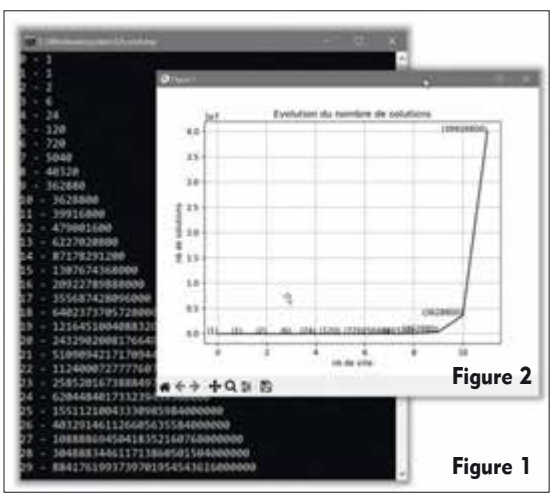


Figure 2

Figure 1

C'est la fameuse croissance exponentielle du nombre de solutions en fonction du nombre de clients, comme on peut le voir sur la **figure 2**, pour 30 villes le nombre de solutions est 30! ce qui représente plus de 10^{31} potentialités.

Un peu de math ?

Nous l'avons compris, une solution est un tableau T et les données sont les distances et pour simplifier, notons la distance entre deux clients j et k.

Soit une tournée d'un problème de TSP avec n sommets. La tournée T se définit par la suite des T_j qui donnent la position j du client dans la tournée. Le coût de la tournée est alors :

$$C(T) = \sum_{j=0}^{n} w_{T_j,T_{j+1}}$$

Une tournée T qui est une solution du problème, est définie par une suite ordonnée de clients. Par exemple, T = [0,2,1,3,0] définit la tournée 0,2,1,3,0.

	0	1	2	3
0	0	1	1.44	2.23
1	1	0	1	1.41
2	1.41	1	0	1
3	2.23	1.41	1	0

Si on prend les distances du tableau, le coût de cette tournée vaut :

$$C(T) = w_{T_0,T_1} + w_{T_1,T_2} + w_{T_2,T_3} + w_{T_3,T_4}$$

$$C(T) = w_{0,2} + w_{2,1} + w_{1,3} + w_{3,0}$$

$$C(T) = 6.11$$

Si on formalise un peu les choses, nous avons quelque chose de simple.

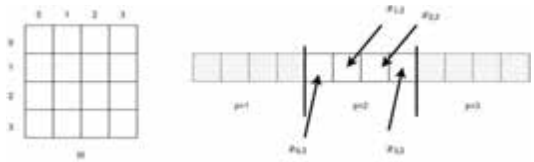
Des données

$W_{i,j}$ La distance entre le client i et le client j

n : nombre de sommets (ici $n = 4$)

Des Variables

$x_{i,p} = 1$ si le client i est affecté à la case p et 0 sinon



Des contraintes

Contrainte 1. À chaque position p est attribuée à un unique client i.

$$\forall p = 1..n-1, \quad \sum_{i=1}^{n} x_{ip} = 1$$

Contrainte 2. Un client i doit apparaître une fois et une seule fois en position p

$$\forall i = 1..n, \quad \sum_{p=1}^{n-1} x_{ip} = 1$$

Contrainte 3. La tournée commence par le client 0 (le dépôt)

$$x_{00} = 1$$

Contrainte 4. La tournée se termine par le client 0 (le dépôt)

$$x_{0n} = 1$$

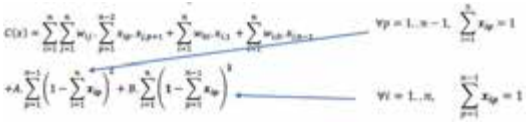
Le coût de la solution est défini par :

$$C(x) = \sum_{i=0}^{n-1}\sum_{j=0}^{n-1} w_{ij} \cdot \sum_{p=1}^{n-2} x_{ip} \cdot x_{j,p+1} + \sum_{i=0}^{n-1} w_{0,i} \cdot x_{i,1} + \sum_{i=0}^{n-1} w_{i,0} \cdot x_{i,n-1}$$

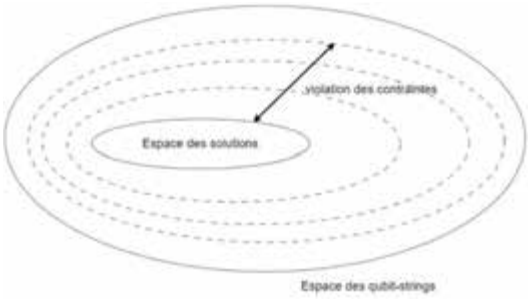
Et le quantique ?

Faire rentrer notre jolie expression mathématique dans une machine quantique pose plusieurs problèmes importants, mais pas insurmontables.

Première mauvaise nouvelle, aucune machine quantique ne sait gérer les contraintes : une machine quantique ne sait traiter que des problèmes sans contrainte. La solution est connue depuis longtemps pour supprimer des contraintes il suffit de les mettre dans la fonction objectif.



Deuxième mauvaise nouvelle, il faut choisir correctement les deux valeurs A et B : si ces deux valeurs sont "trop" grandes, la partie de $C(x)$ qui modélise les contraintes va "écraser" les termes qui représentent le coût de la solution.



Troisième mauvaise nouvelle, malgré tous nos efforts, il est possible que l'on génère des tableaux T qui ne respectent pas les contraintes.

En un mot : on n'est pas exactement au bon endroit dans notre paysage ! ☹

2 - Le quantique vient à notre secours

Les portes viennent à nous

Les portes quantiques vont être à notre machine quantique ce que les variables sont à notre modèle mathématique.

Une variable x se traduire en porte $\frac{1}{2}(Id - Z)$ qui combine à la fois $Id = \begin{pmatrix} 1 & \square \\ \square & 1 \end{pmatrix}$ la matrice identité et par $Z = \begin{pmatrix} 1 & \square \\ \square & -1 \end{pmatrix}$.

x est une variable binaire et prenant deux valeurs possibles qui sont 0 et 1. Si on regarde attentivement la matrice de

$$\frac{1}{2}(Id - Z) = \begin{pmatrix} 0 & \square \\ \square & 1 \end{pmatrix}$$

on voit sur la diagonale un 0 et une valeur 1.

$$Z = \begin{pmatrix} 1 & \square \\ \square & -1 \end{pmatrix}$$

$$\frac{1}{2}(Id - Z) = \begin{pmatrix} 0 & \square \\ \square & 1 \end{pmatrix}$$

$$\frac{1}{2}(Id + Z) = \begin{pmatrix} 1 & \square \\ \square & 0 \end{pmatrix}$$

Ce qu'on lit sur la diagonale des matrices correspond en un sens à la variable x. Ainsi remplacer x par $\frac{1}{2}(Id - Z)$ va conduire à une nouvelle expression composée exclusivement de matrices qui vont former ce qu'on appelle un Hamiltonien.

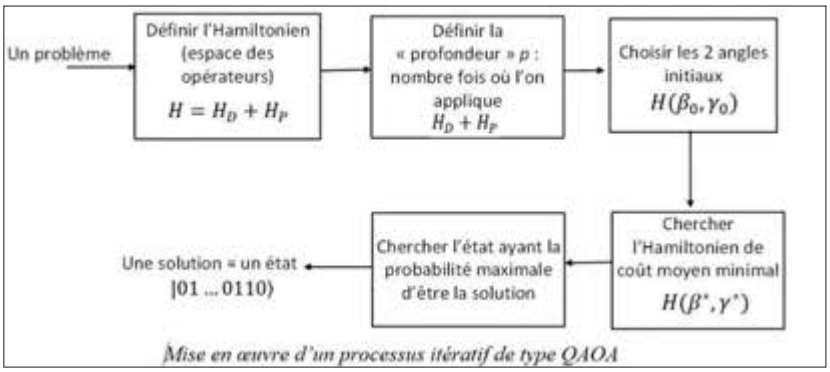
Évaluer un Hamiltonien

Un Hamiltonien H permet de modéliser une distribution de probabilités sur l'ensemble des solutions et il dépend de paramètres de sorte qu'on écrit généralement $H(\alpha, \beta)$ où α et β sont deux paramètres et plus précisément des angles.

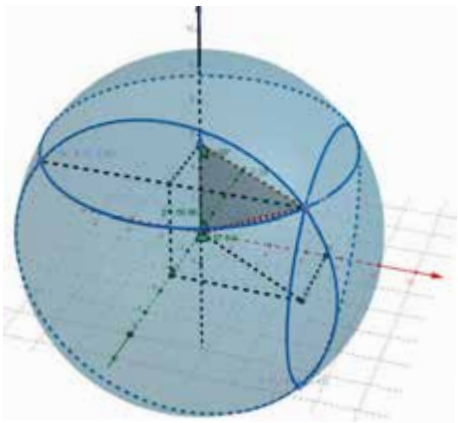
La méthode la plus connue est l'optimisation adiabatique et elle consiste à faire un "mélange" entre H_D un hamiltonien initial qui modélise le fait qu'on ne sait rien, on ne sait pas où se trouve les bonnes solutions (c'est donc une distribution uniforme) et H_p un Hamiltonien qui modélise le problème qu'on cherche à résoudre.

$$H(t) = s(t).H_D + [1 - s(t)].H_P$$

Dans le cas particulier de **l'optimisation adiabatique**, il suffit de choisir : $s(t) = 1 - t$. et Δt où $\Delta t = \frac{1}{n}$ est le nombre de pas utilisés pour "passer" de H_D à H_p. L'inconvénient : choisir Δt petit.



Mise en œuvre d'un processus itératif de type QAOA



Avec la méthode QAOA, on procède différemment : on cherche les deux paramètres α et β qui donnent "une bonne" distribution de probabilités, c'est-à-dire une distribution où les solutions de coûts faibles ont des probabilités élevées.

Il faut se souvenir, comme le montre le schéma à côté qu'un état "quantique" peut se définir comme une configuration particulière.

3 - Comment coder un algorithme de type QAOA

La définition de QAOA passe par la définition des fonctions suivantes :

- Une fonction python **H_P (gamma…)** qui reçoit en paramètre un tableau contenant les angles et qui génère le circuit associé à H_p. (Le problème)
- Une fonction python **H_D (gamma…)** qui reçoit aussi en paramètre un tableau contenant des angles et qui génère le circuit associé à H_d (le mixeur).
- Une fonction python **H (beta, gamma, profondeur)** qui génère le circuit qui se compose d'une succession de H_p, H_d.
- Une fonction python **valeur_objectif (une_solution)** qui à chaque solution représentée sous forme binaire (100101 par exemple) associe la valeur de la fonction objectif.
- Une fonction python **evaluer_H(counts)** qui fournit $C^p(\vec{\beta}, \vec{\gamma})$: le coût moyen de la distribution donnée dans le

Abonnez-vous à programmez!

Formules	1 an	2 ans	Etudiant	Numérique
Vous recevez le magazine chez vous	1 an 10 numéros (papier) 55 €	2 ans 20 numéros (papier) 90 €	1 an 10 numéros (papier) 45 €	1 an 10 numéros (format PDF) 45 €
Abonnements + accès aux archives	80 € (1 an + archives)	120 € (2 ans + archives)		70 € (1 an + archives)

Tarifs France métropolitaine.
Tarif PDF : valable partout dans le monde

L'abonnement comprend : les numéros normaux et les hors-séries

L'abonnement **numérique** est disponible uniquement sur notre boutique : **www.programmez.com**

Oui, je m'abonne

ABONNEMENT à retourner avec votre règlement à :
PROGRAMMEZ, Service Abonnements
57 Rue de Gisors, 95300 Pontoise

☐ Abonnement 1 an :	55 €
☐ Abonnement 2 ans :	90 €
☐ Abonnement 1 an Etudiant : Photocopie de la carte d'étudiant à joindre	45 €

Option : accès aux archives

☐ Pour abonnement 1 an	25 €
☐ Pour abonnement 2 ans	30 €

☐ Mme ☐ M. Entreprise : Fonction :

Prénom : Nom :

Adresse :

Code postal : Ville :

Adresse email indispensable pour la gestion de votre abonnement

E-mail : @

☐ Je joins mon règlement par chèque à l'ordre de Programmez !

☐ Je souhaite régler à réception de facture

* Tarifs France métropolitaine

Offres pouvant s'arrêter à tout moment, sans préavis **PROG 270**

Boutique ÉtÉ Boutique ÉtÉ Boutiq

Les anciens numéros de programmez!

Le magazine des dév - CTO - Tech Lead

260

264

266

267

268

HS18

269

Complétez votre collection.

Tous les numéros sont disponibles en PDF

Tarif unitaire 8 € (6,99€ + 1,01€) (frais postaux inclus)

☐ *260 uniquement en pdf*
☐ *264 uniquement en pdf*
☐ *267 uniquement en pdf*
☐ *266* : |__| ex
☐ *268* : |__| ex
☐ *HS18* : |__| ex
☐ *269* : |__| ex

Commande à envoyer à :
Programmez!
57 rue de Gisors
95300 Pontoise

soit |__|__| exemplaires x 8 € = |__|__|__|__| € .....

soit au **TOTAL** = |__|__|__|__|__| €

☐ M. ☐ Mme Entreprise : |__|__|__|__|__|__|__|__| Fonction : |__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|

Prénom : |__|__|__|__|__|__|__|__|__|__|__|__| Nom : |__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|

Adresse : |__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|

Code postal : |__|__|__|__|__| Ville : |__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|

Règlement par chèque à l'ordre de Programmez ! I Disponible sur www.programmez.com

Offres pouvant s'arrêter à tout moment, sans préavis **PROG 270**

dictionnaire **counts**. Ce dictionnaire est de la forme (une_solution : n_occurence).

D'un point de vue purement technique il est simple en Python de créer une fonction qui permette de générer une fonction avec les "bons" paramètres. Notez que *f(theta)* la fonction créée utilisé la fonction H qui représente l'Hamiltonien global.

```
def definir_la_fonction_a_minimiser(p):

    def f (theta):
        print(theta)
        beta  = theta[:p]
        gamma = theta[p:]
        qc = H (beta, gamma, p)

        qc_compiled = transpile(qc, bac
        job_sim = backend.run(qc_compiled, shots=NUM_SHOTS)
        result_sim = job_sim.result()
        counts = result_sim.get_counts(qc_compiled)

        les_resultats = inversion_affichage(counts)
        res = evaluer_H(les_resultats)
        return res

    return f
```

Le point clé : la fonction qui décrit l'Hamiltonien

Le circuit avec ses qubits

```
def H (beta, gamma, profondeur):
    qreg_q = QuantumRegister(nb_dequbits, 'q')
    creg_c = ClassicalRegister(nb_dequbits-1, 'c')
    qc = QuantumCircuit(qreg_q, creg_c)

    # profondeur du circuit : p = len(beta)
    p = profondeur
    # initialisation

    for i in range (0,p):
        qcb = H_D (beta[i])
        qc.append(qcb, [12] + [i for i in range(0
        qc.barrier()

        qc_news = H_P (gamma[i])
        qc.append(qc_news, [i for i in range(0,nb_dequbits) ] )
        qc.barrier()

    # on fait la mesure
    for i in range(nb_dequbits-1) :
        qc.measure(qreg_q[i], creg_c[i])

    return qc
```

Le H_D dont nous avons parlé précédemment.

Le H_p dont nous avons parlé précédemment et qui est l'Hamiltonien du TSP

Eric Bourreau, Gérard Fleury, Philippe Lacomme
Introduction à l'informatique quantique
Eyrolles Blanche 6 Décembre 2022
ISBN: 9782416006531

Gérard Fleury, Philippe Lacomme
Les algorithmes de base de l'informatique quantique - Tome 2
Eyrolles Blanche 2023
ISBN: 9782212772784

Pour résoudre Le cœur de l'algorithme choisir p qui permet de choisir le nombre d'angles α et β qu'on va utiliser et créer la fonction objectif à minimiser.

```
p = 4
la_fonction_objectif = definir_la_fonction_a_minimiser(p)
```

Comme pour un problème de minimisation classique, il faut choisir un point de départ c'est-à-dire ici des angles initiaux et initialiser le processus.

```
init_point = np.array([1, 1, 1, 1,1, 1, 1, 1])
res = la_fonction_objectif (init_point)
```

Pour optimiser c'est-à-dire trouver les "bons" angles on peut utiliser les facilités Python via la méthode minimize(). Cette méthode reçoit en paramètre les points angle initiaux (stockés ici dans la variable init_point) et une méthode d'optimisation…. Il existe plusieurs méthodes que vous pouvez trouver dans la documentation de scipy.

https://docs.scipy.org/doc/scipy/reference/optimize.html

Attention, elles ne se valent pas toutes et vous devrez peut-être en tester plusieurs. Celle qu'on retrouve le plus souvent est la méthode Cobyla.

```
res_sample = minimize(la_fonction_objectif,
                      init_point,
                      method='Cobyla',
                      options={'maxiter':50, 'disp': True}

True}


                      )
print(res_sample)
```

Il ne reste plus qu'à afficher la solution…

```
print("\n meilleur hamiltonien \n")
 optimal_theta = res_sample['x']
beta  = optimal_theta[:p]
gamma = optimal_theta[p:]

print("les angles trouves : " , optimal_theta)
print("les angles beta : ", beta)
print("les angles gamma : " , gamma)
```

En utilisant ces paramètres, on peut appeler une dernière fois le circuit pour obtenir la distribution finale.

4 - Comment tester des codes quantiques

Il vous suffit de consulter la page
https://perso.isima.fr/~lacomme/Quantique/index.php
c'est la "companion web page" du livre " Introduction à l'informatique quantique" qui est paru en 2022 aux éditions Eyrolles.

5 - Quelques essais

On peut facilement vérifier que pour le problème simple de départ, nous n'avons que 6 solutions. Deux solutions parmi les 6 donnent le coût optimal de 4.82.

Pour savoir si la méthode fonctionne : rien de plus simple. Il faut tester avec différents paramètres. Rapidement avec 200

Les différentes solutions du TSP

Solution	Coût
100001010	4.82
010100001	6.05
001010100	5.23
100010001	5.23
010001100	4.82
001100010	6.05

itérations de recherche on obtient des distributions de probabilités où 15% de la distribution se concentre sur les deux solutions qui correspondent à 4.82.

Conclusion

Cette petite introduction devrait vous donner envie d'en savoir plus. Plusieurs dessins et schémas viennent des deux livres cités qui peuvent vous servir d'introduction : ils sont accessibles à tous ceux qui possèdent des bases de mathématiques.

Le Cloud de Quandela :
l'informatique quantique pour tous, dès aujourd'hui !

L'offre Cloud de Quandela permet à tout développeur d'accéder instantanément à un véritable ordinateur quantique photonique via une simple connexion Internet. Explorez la puissance des algorithmes quantiques basés sur la lumière - sans laboratoire ni équipement spécialisé.

Samuel Horsch
Quantum Applications Architect, Quandela

Le Cloud de Quandela est une véritable passerelle vers les ordinateurs quantiques photoniques de Quandela. Un tableau de bord unique permet de gérer son compte, d'accéder à la « Quantum Toolbox » (boîte à outils d'algorithmes quantiques) et de suivre l'état de ses tâches. Les tâches sont exécutées à l'aide de la bibliothèque Python Perceval, qui se connecte directement avec les QPU et les plateformes de simulation. Qu'il s'agisse de créer des prototypes de nouveaux circuits photoniques ou de développer des algorithmes quantiques sur mesure, le Cloud de Quandela apporte des ressources photoniques directement dans l'éditeur de code. Il permet également de réaliser des applications scientifiques de pointe, le tout à partir d'une interface fluide et parfaitement intégrée, accessible via le navigateur. **Figure 1**

Pourquoi des processeurs quantiques photoniques ?

Les processeurs quantiques photoniques permettent l'exécution d'algorithmes photoniques natifs pouvant dépasser les architectures classiques dans des applications spécifiques. L'architecture photonique présente des avantages distinctifs de l'approche photonique :

- Fonctionnement à température ambiante : Contrairement aux processeurs quantiques supraconducteurs qui nécessitent des températures proches du zéro absolu, les QPU photoniques opèrent dans des conditions normales
- **Absence de** décohérence : Les photons n'interagissent pas avec le monde extérieur, préservant ainsi parfaitement leurs propriétés quantiques sans dégradation
- Parallélisme naturel : La nature ondulatoire de la lumière permet d'explorer simultanément de multiples solutions
- Évolutivité : L'architecture photonique se prête mieux à la montée en échelle vers des systèmes plus complexes

Comment accéder à cette technologie aujourd'hui ? La réponse se trouve dans Perceval - une bibliothèque Python qui permet l'accès direct au Cloud de Quandela.

Perceval

Perceval est une bibliothèque Python open source développée par Quandela. L'installation se fait simplement avec ***pip install perceval-quandela***. Pour tout développeur Python, la prise en main est rapide et intuitive.

Perceval intègre des outils de conception et de simulation de circuits pour itérer et déboguer rapidement. Les simulations classiques fonctionnent jusqu'à 30-40 photons, au-delà les processeurs quantiques révèlent tout leur potentiel.

La manière la plus simple de se familiariser avec un nouvel outil de programmation est d'utiliser un exemple de type *« Hello World »* : ***Hello Quantum World***

Avant de commencer à écrire le code, voici un bref aperçu de la technologie quantique de Quandela. Nous utilisons des photons uniques (particules de lumière) envoyés à travers une puce photonique dédiée aux calculs complexes. Cela permet de détecter les photons et de lire les résultats.

La puce photonique contient plusieurs canaux optiques, similaires aux fibres optiques qui transportent Internet. **Figure 2**

La puce intègre deux composants optiques essentiels pour construire des circuits quantiques : le déphaseur (phase shifter) et le séparateur de faisceau (beam splitter). **Figure 3**

Prenons à présent un exemple simple comme celui de *« Hello Quantum World »* :

Figure 2 :
Schéma des canaux optiques représentés par des lignes noires et des photons par des points rouges.

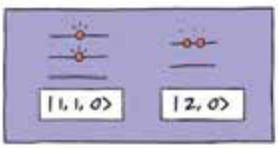


Figure 3 :
Schéma des deux composants optiques, le déphaseur et le séparateur de faisceau.

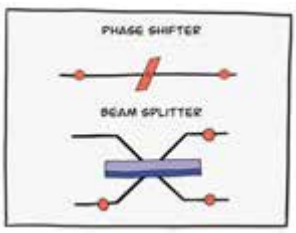
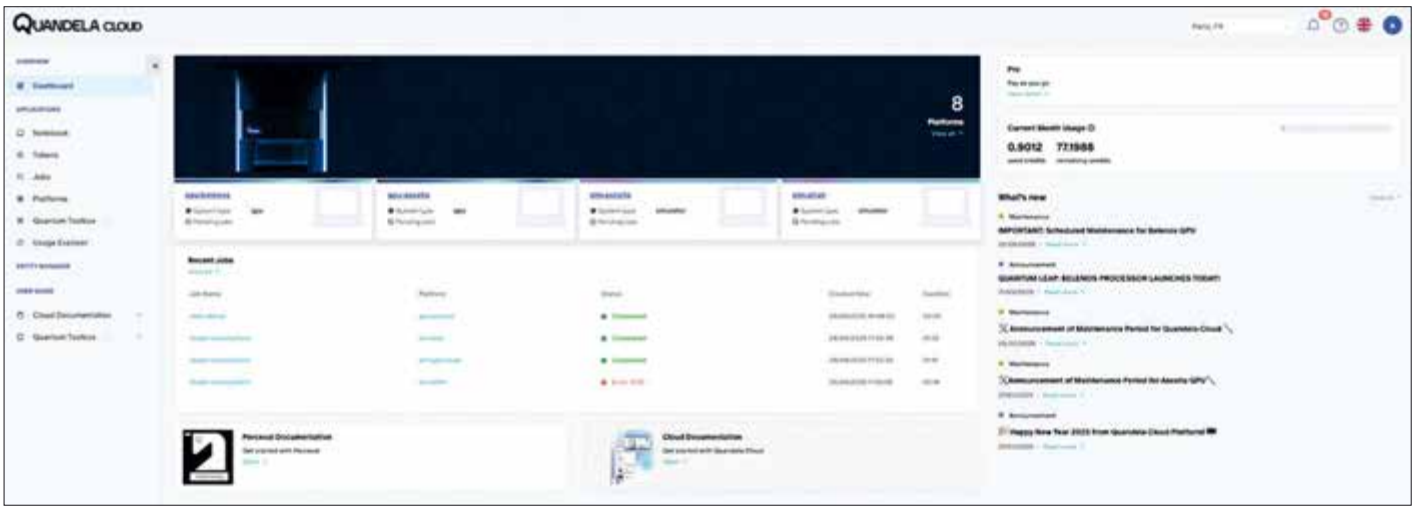


Figure 1



1 Créer un état d'entrée photonique

Dans cet exemple, nous préparons un photon dans chacun des deux canaux optiques :

```
input_state = pcvl.BasicState("|1,1>")
```

2 Conception du circuit

Nous utiliserons un circuit simple avec un seul séparateur de faisceau :

```
circuit = pcvl.Circuit(2) // pcvl.BS()
```

3 Se connecter au Cloud de Quandela

Créez une instance de processeur à distance en spécifiant une plateforme (par exemple, « qpu:ascella ») et en saisissant votre *token*personnel à partir de votre compte sur le Cloud de Quandela

```
remote_processor = pcvl.RemoteProcessor("qpu:ascella", token="your token")
```

4 Effectuer la tâche

Définissez le circuit, l'état d'entrée et les filtres de détection. Soumettez ensuite le travail :

```
remote_processor.set_circuit(circuit)
remote_processor.min_detected_photons_filter(2)
remote_processor.with_input(input_state)
remote_sampler = pcvl.algorithm.Sampler(remote_processor,
                        max_shots_per_call=1e12)
remote_sampler.default_job_name = "Hello World"
remote_job = remote_sampler.sample_count.execute_async(1e3)
```

Ce code soumettra une tâche à l'ordinateur quantique de Quandela via votre compte cloud. Une fois la tâche terminée, vous pourrez consulter les résultats.

```
# Attendre la fin de l'exécution et récupérer les résultats
results = remote_job.get_results()
# Afficher les résultats
print("Résultats de l'échantillonnage quantique :")
for state, count in results.items():
  probability = count / 1000 # sur 1000 échantillons
  print(f"État {state}: {count} occurrences ({probability:.1%})")
```

Quel est le résultat attendu ? Pour cet exemple simple, la sortie montrera une probabilité de 50% d'identification de la paire de photons dans l'état |0,2> et 50% dans |2,0>.

Nature probabiliste des résultats quantiques :

Contrairement aux ordinateurs classiques qui produisent des résultats déterministes, les ordinateurs quantiques génèrent des distributions de probabilités. Chaque exécution peut donner un résultat légèrement différent, c'est pourquoi nous répétons les mesures (ici 1000 fois) pour estimer les probabilités de chaque état de sortie. Cette nature probabiliste n'est pas une limitation, mais une caractéristique fondamentale qui permet aux algorithmes quantiques de résoudre certains problèmes plus efficacement que les approches classiques.

Algorithmes quantiques

À présent que vous savez comment vous connecter au cloud et exécuter un circuit quantique, explorez maintenant les différents types d'algorithmes. La boîte à outils quantique propose cinq algorithmes "prêts à l'emploi", accessibles directement depuis le tableau de bord du cloud ou via votre éditeur de code :

Chimie VQE

Le Variational Quantum Eigensolver (VQE) est un algorithme quantique bien connu qui permet de calculer les énergies des états fondamentaux. L'algorithme Chemistry VQE de Quandela permet aux utilisateurs d'explorer des molécules telles que H_2, LiH ou BeH_2. Cet outil de démonstration est idéal pour l'apprentissage et l'expérimentation.

VQE personnalisé

Cette version de VQE permet l'accès à n'importe quel Hamiltonien, favorisant des applications au-delà de la chimie moléculaire. C'est un outil flexible pour développer de nouveaux cas d'utilisation.

CVar-VQE

Abréviation de Conditional Value at Risk VQE, cette variante est conçue pour résoudre des problèmes d'optimisation exprimés dans un format QUBO (Quadratic Unconstrained Binary Optimisation). Il suffit de fournir une matrice QUBO pour que l'algorithme renvoie une solution optimale.

Isomorphismes de graphes

Cet algorithme photonique natif détermine si deux graphes sont structurellement identiques (isomorphes). L'algorithme compare les deux graphes et indique *True* s'ils sont isomorphes, ou *False* s'ils ne le sont pas.

Identification des sous-graphes denses

En utilisant des méthodes similaires à celles de l'algorithme d'isomorphisme des graphes, cet outil identifie les sous-graphes denses qui sont des sous-ensembles d'un graphe muni d'un grand nombre de connexions - à l'intérieur d'un seul graphe en entrée.

En plus des algorithmes de la boîte à outils quantique, vous pouvez développer de nouveaux algorithmes ou adapter des algorithmes existants à de nouveaux cas d'utilisation. Dans la documentation de Perceval se trouvent de nombreux exemples de démonstrations dont on peut s'inspirer. L'équipe Algorithmes de Quandela publie tous ses résultats de recherche, afin que les utilisateurs puissent les étudier.

Comment rejoindre la communauté

Quandela soutient une communauté de développeurs quantiques active et en pleine croissance. Les ressources disponibles pour développer ses compétences sont les suivantes:

- « Training Center » Quandela : Des parcours de formation axés sur les algorithmes photoniques et la technologie de Quandela sont proposés régulièrement
- Forum des développeurs : Poser des questions, partager ses idées et bénéficier du soutien de la communauté de développeurs
- Source ouverte sur GitHub : Contribuer à Perceval et explorer le code source et les différentes mises à jour.

Que vous soyez un développeur découvrant l'informatique quantique pour la première fois ou un expert des circuits quantiques complexes, le Cloud de Quandela et Perceval offrent un accès inédit dans le monde de l'informatique quantique photonique. L'avenir photonique est plus proche que vous ne le pensez.

Un exemple d'une classe d'algorithmes quantiques : VQE (Variational Quantum Eigensolver)

Jean-Michel TORRES

Passionné d'informatique quantique,

Enseigne le calcul quantique dans de nombreuses écoles et universités.

Depuis le milieu des années 2010, une nouvelle classe d'algorithmes quantiques intéresse beaucoup la communauté du calcul quantique, elle pourrait en effet permettre d'atteindre le fameux avantage quantique, avant de disposer d'ordinateurs quantiques capables de correction d'erreur. Il s'agit d'algorithmes quantiques variationnels (VQA Variationnal Quantum Algorithms), voyons un exemple : VQE.

Les machines, mais pas seulement.

Les annonces des constructeurs d'ordinateurs quantiques se succèdent et sont de plus en plus impressionnantes. Cela révèle sans doute que le moment où l'ordinateur quantique atteindra clairement l'étape majeure de l'**avantage** quantique est très proche (c'est le moment où un ordinateur quantique sera capable de fournir un résultat de calcul utile hors d'atteinte de nos super calculateurs classiques).

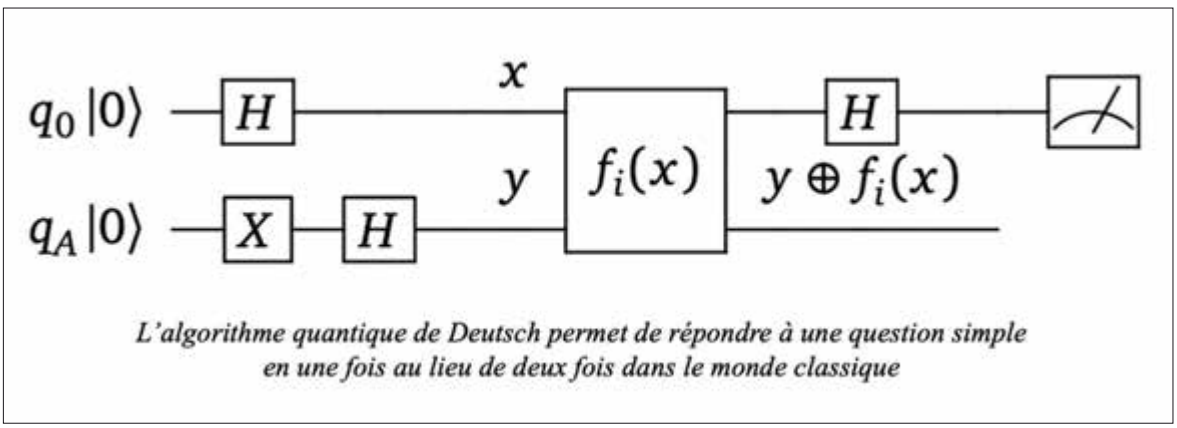
Le calcul quantique présente de nombreux défis : maîtriser la physique quantique expérimentale pour mettre en œuvre des qubits de très bonne qualité, en nombre suffisant pour parvenir à exécuter des calculs permettant l'avantage quantique. Et dans ce domaine, même l'avenir promet d'importants progrès, il existe des machines qui ont déjà la possibilité en théorie de fournir cet avantage quantique (c'est une thèse discutée par John Preskill dès 2018 : Quantum Computing in the NISQ area and beyond (arXiv 1801.00862).

Mais pour cela, il y a d'autres défis : trouver les cas de calcul les plus favorables pour démontrer cet avantage rapidement et trouver l'algorithme qui le fera.

Les résultats en algorithmie quantique ont précédé la conception des ordinateurs quantiques, et peut-être même ont-ils motivé les constructeurs d'ordinateurs quantiques à continuer leurs efforts.
En effet les premiers concepts d'algorithmes datent des années 1980 alors même que l'avenir du calcul quantique était vraiment incertain, et qu'il n'existait aucune machine quantique, ne serait-ce qu'à un seul qubit. **Figure 1**

En 1985 David Deutsch prouve que l'on peut obtenir avec le principe d'intrication quantique en une seule étape un résultat qui nécessite toujours deux étapes dans un raisonnement classique. Il s'ensuit une série d'inventions d'algorithmes quantiques : les algorithmes de Deutsch-Josza, Bernstein-Varizani, Simon, toujours plus impressionnants dans leur ingéniosité, et dans leur avantage en termes de complexité algorithmique. Par exemple l'algorithme de Simon possède un avantage en temps exponentiel par rapport à la résolution classique du problème associé, mais ces algorithmes n'ont aucune application pratique.
Dans les années 1990, toujours pas d'ordinateur quantique, mais deux algorithmes marquent l'histoire : l'algorithme de Deustch (recherche plus efficace dans une liste non ordonnée) et celui de Shor (factorisation de nombre entier). Ces deux algorithmes présentent le gros avantage de résoudre des problèmes « utiles », mais en même temps le gros désavantage de ne pas pouvoir être exécutés de manière efficace sur les ordinateurs quantiques actuellement disponibles. Pour envisager cela, il faudra disposer d'ordinateurs quantiques mettant en œuvre les techniques de correction d'erreur (qui sont beaucoup plus complexes que la détection et la correction d'erreurs de parités bien connues sur les ordinateurs classiques).

Figure 1



L'algorithme quantique de Deutsch permet de répondre à une question simple en une fois au lieu de deux fois dans le monde classique

Il faut donc trouver autre chose

L'idée est d'utiliser une autre approche du calcul et si possible une approche particulièrement favorable aux qubits et à leur domaine de prédilection : l'algèbre linéaire.
C'est une approche riche de promesses, mais beaucoup plus complexe, et c'est d'ailleurs seulement une approche non classique de la résolution d'un problème qui permet au calcul quantique d'être pertinent. La raison est que l'ordinateur quantique apporte superposition et intrication, et si ces propriétés ne sont pas exploitées, il n'y a pas de raison d'espérer grand-chose du calcul quantique.

Une des possibilités est donc la famille des algorithmes quantiques variationnels : VQA, et l'instance la plus simple à décrire ici est VQE, Variational Quantum Eigensolver, en français : solveur quantique variationnel de valeurs propres.

Avant de commencer : je ne cherche pas à exposer la totalité de la construction mathématique : je me contente de présenter le raisonnement de VQE sans entrer dans les détails, sans exposer les démonstrations ni les calculs, cependant je vais devoir faire appel à des notions ou des termes que vous avez peut-être déjà oubliés.

Je m'explique sur les trois initiales de VQE :

V : C'est un algorithme variationnel : c'est-à-dire qu'il s'agit de faire varier un certain nombre de paramètres pour trouver le résultat recherché. La « variation » va consister à faire des itérations, pilotées par un mécanisme « classique » : tout simplement un optimiseur. Il en existe un certain nombre, et c'est un principe déjà répandu, dans le monde de l'optimisation (recherche opérationnelle) et par exemple en apprentissage automatique, lorsque l'on cherche un minimum (changement des poids interneuronaux, en étapes (« epoch ») jusqu'à atteindre un fonctionnement optimal (« loss fonction »). On utilisera donc une condition d'arrêt (lorsque la valeur obtenue ne varie plus « beaucoup », on arrête, considérant que le résultat est atteint. Ce mécanisme est confié à un algorithme d'optimisation classique.

Q : En plus d'être variationnel il est quantique : il va y avoir des itérations entre l'optimiseur classique (qui ajuste les paramètres à chaque tour, les présente au calculateur quantique, reçoit le résultat du calcul quantique et ajuste à nouveau les paramètres « dans le bon sens » (c'est le job de l'optimiseur) pour le prochain tour. On détaille plus bas ce que l'on attend de l'ordinateur quantique.

E : Et pour finir, il trouve une valeur propre (« Eigenvalue »).
Qu'est-ce qu'une valeur propre ? C'est une notion d'algèbre linéaire, étant donné un opérateur (une matrice M), un vecteur propre u (associé à une valeur propre λ) est tel que $Mu = \lambda u$
Dès que la matrice est d'assez grande taille, plusieurs centaines ou milliers de lignes, il est très difficile (au sens de la complexité algorithmique) de trouver une valeur propre. C'est un calcul qui peut prendre un temps très long (déraisonnable par rapport aux résultats du calcul : s'il faut plus d'un jour pour agencer de manière optimale les colis dans un camion par exemple)

À quoi sert une valeur propre ? En physique, par exemple dans l'étude des molécules en chimie, on peut représenter les propriétés de la molécule dans une matrice, appelée Hamiltonien. Il se trouve que cette matrice est hermitienne (propriété mathématique qui lui confère de nombreuses propriétés parmi lesquelles celle d'avoir les valeurs propres réelles), le lien se fait ici : les valeurs propres de l'Hamiltonien (réelles, donc) correspondent (pour des raisons liées aux postulats de la mécanique quantique) au niveau d'énergies possibles pour l'objet en question (la molécule par exemple).
En particulier VQE « promet » de trouver la valeur propre minimale, donc le niveau d'énergie fondamental de la molécule étudiée (que les chimistes me pardonnent ceci est une simplification grossière).

Et maintenant l'algorithme

On a fait le plus dur, encore un petit effort, avec un minimum de connaissances en physique quantique ou en algorithmes quantiques, ça devrait aller :

1 Expectation value : souvenons-nous que l'on note $|\psi\rangle$ le vecteur représentant un système quantique. (le premier postulat de la mécanique quantique nous dit qu'à un système quantique est associé un espace vectoriel E sur $\mathbb{C}$, dans lequel les états sont définis par les vecteurs de norme 1 de cet espace.
L'Hamiltonien qui représente la physique du phénomène est une matrice H.
On définit l'expectation value par cette formule : $\langle \psi | H | \psi \rangle$, on se souvient que le produit d'un matrice par un vecteur ($H|\psi\rangle = |\psi'\rangle$) est un vecteur (représentant l'état résultant de la transformée de $|\psi\rangle$ par H) et puis $\langle\psi|$ est le vecteur « dual » (transposée conjuguée de $|\psi\rangle$), et donc $\langle \psi | H | \psi \rangle = \langle \psi | \psi' \rangle$ est un produit scalaire* des deux vecteurs (* produit hermitien pour les puristes), c'est donc un nombre (réel comme indiqué plus haut), il se trouve que la machine quantique sait calculer efficacement cette quantité (par exemple avec qiskit il s'agit de la fonction d'estimation)

2 Imaginons que p désigne un ensemble de paramètres (degrés de liberté) et que l'on utilise les paramètres p pour générer des états $|\psi(p)\rangle$, et que l'on dispose d'une méthode pour faire varier p pour parcourir « intelligemment » l'espace des états possibles du système.

3 Il existe de nombreuses publications qui définissent ces méthodes et la manière de les transformer en circuits quantiques pour l'ordinateur quantique auquel on fournit les paramètres p, et qui construit l'état $|\psi(p)\rangle$.

4 Un théorème (relativement facile à prouver) montre que cette valeur $\langle \psi(p) | H | \psi(p) \rangle$, est supérieure ou égale à la valeur propre minimale de H

5 Admettons (ça se démontre également) que l'on peut « facilement » construire un circuit quantique qui applique H (c'est-à-dire sans nécessiter un nombre d'opérations quantiques qui croît trop rapidement),

Le tour est joué, en regroupant ces 5 éléments, on peut représenter la démarche VQE avec ce schéma : **Figure 2**

L'algorithme se décrit comme suit :

1 - Choix des paramètres de départ (p_0) et de la méthode construction de $|\psi(p)\rangle$
2 - Calcul (quantique) de $|\psi(p)\rangle$, et de l'Expectation Value
3 - Ajustement des paramètres p par l'optimiseur classique
4 - Répéter 2 et 3 jusqu'à satisfaire les critères de convergence (connus de l'optimiseur)
5 - Fin : on a obtenu une valeur approchée de la valeur propre minimale.

Un tout petit exemple pour illustrer ceci pour ceux qui apprécient le calcul : **Figure 3**

Et pour finir une remarque très importante :
Au-delà des calculs d'énergie pour les molécules en chimie, la communauté de la recherche opérationnelle a imaginé des méthodes permettant de représenter une fonction à minimiser sous la forme d'une matrice hermitienne, il ne reste plus qu'à utiliser VQE pour trouver la valeur minimale !

Convaincus ?

Avec les mots clefs **qiskit** et **VQE** sur un moteur de recherche, vous devriez arriver sur un notebook Jupyter qui explique cette démarche VQE vous permet avec quelques lignes de Python de tourner l'algorithme VQE pour deux problèmes « concrets » : un MAXCUT (coupure maximale des sommets d'un graphe) et TSP (traveller sales person) pour trouver le plus court chemin... vers le calcul quantique.

J'espère avoir clarifié les principes de VQE et surtout de vous avoir donné l'envie d'approfondir les détails pour essayer vous-même.

Figure 2

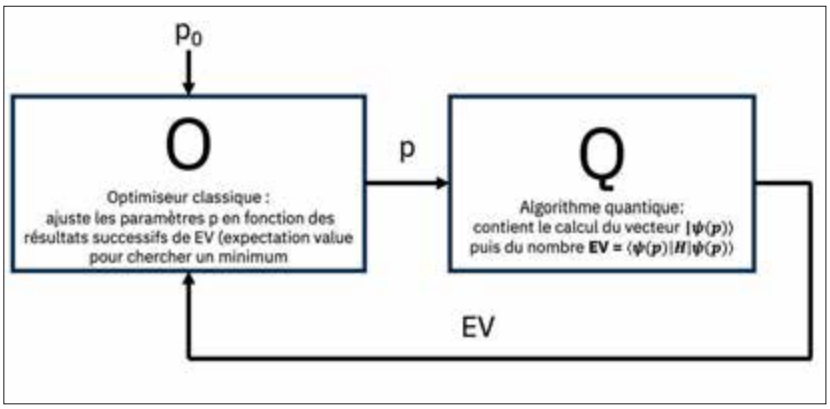


Figure 3

L'opérateur de Pauli Z est hermitien, ses vecteurs propres : $|0\rangle$ (valeur propre 1) et $|1\rangle$ (valeur propre -1), il s'agit justement des deux observables énergie du qubit, Z correspond à l'Hamiltonien du qubit dans ces conditions.

$$Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

L'état d'un qubit peut s'écrire sous cette forme générale :

$$|\psi\rangle = cos\frac{\theta}{2}|0\rangle + e^{i\phi}sin\frac{\theta}{2}|1\rangle \text{ et donc } \langle\psi| = cos\frac{\theta}{2}\langle 0| + e^{-i\phi}sin\frac{\theta}{2}\langle 1|$$

Calcul de $\langle\psi|Z|\psi\rangle$:

$$Z|\psi\rangle = cos\frac{\theta}{2}|0\rangle - e^{i\phi}sin\frac{\theta}{2}|1\rangle$$

$$\langle\psi|Z|\psi\rangle == \left(cos\frac{\theta}{2}\langle 0| + e^{-i\phi}sin\frac{\theta}{2}\langle 1|\right) \times \left(cos\frac{\theta}{2}|0\rangle - e^{i\phi}sin\frac{\theta}{2}|1\rangle\right)$$

$$\langle\psi|Z|\psi\rangle = cos^2\frac{\theta}{2} - sin^2\frac{\theta}{2}$$

Dans le contexte de VQE : l'angle θ est le paramètre p (paramètre unique, en général p est un ensemble de paramètres). Lorsqu'on fait varier θ entre 0 et 2π l'EV varie de 1 à -1 et on trouve ainsi la valeur propre minimale pour $\theta = \pi$, et elle vaut donc-1.

Benjamin GIGON
Manager Référent – OCTO Technology

Sécurité applicative : le détournement des fonctions internes d'un programme

Lors d'une mission, un client souhaitait vérifier la sécurité de son logiciel. Son logiciel utilisait fortement les méthodes cryptographiques afin de délivrer un service particulier à ses clients. Son logiciel était déployé sur une multitude de serveurs hébergés par les clients eux-mêmes, ils avaient donc accès à son logiciel sans contrainte particulière. Ne voulant pas trop rentrer dans les détails, les données manipulées étaient critiques et une possibilité de déchiffrer les données en dehors du workflow prévu ou de pouvoir récupérer des données sensibles (comme des clefs de chiffrements ou des certificats privés) était considérée comme une faille majeure et une perte de certification de l'autorité de certifications.

Le but du client était de savoir si son logiciel était assez sécurisé pour ne pas compromettre la sécurité globale de toute son architecture logicielle. Pour cela, le défi était de voir s'il était possible d'extraire toutes informations sensibles ou cryptographiques venant de son logiciel (données non chiffrées, clefs de chiffrement, certificats privés, etc...)
Après une (très) brève recherche, nous avons pu mettre en évidence des **techniques de contournements** permettant d'extraire des données sensibles, des clefs et des certificats privés sans toucher à l'applicatif ni même en étant intrusif et sans laisser de trace.
La méthode présentée ici est un résumé (et anonymisée) d'une des techniques utilisées lors de cette mission.
Les outputs ont été modifiés pour être lisibles.

Généralité sur les fonctions

Pour faire simple, les applications sont découpées dans de multiples tâches internes appelées fonctions. Ces fonctions sont pour la plupart à l'intérieur même du projet. Ainsi, si vous faites :

```
  (...)
void display(void) {
  printf("Hello World !\n");
}
(...)
int main(void) {
  display();
}
```

Votre fonction **display** fait partie même de votre projet, vous l'avez codé et vous connaissez sa structure interne.
A contrario, la fonction **printf** ne fait partie de votre projet, elle fait partie d'une bibliothèque externe et sera intégrée (directement ou indirectement) à votre programme lors de la phase de compilation. **printf** fait partie - pour notre cas - de la glibc.
Lors d'une compilation, vous avez le choix entre une liaison dynamique ou une liaison statique avec ces fonctions "externes".

- La liaison dynamique va laisser la "définition" des fonctions externes dans leurs bibliothèques associées. Par exemple, pour **printf**, cette dernière sera dans la glibc donc **libc.so**. On utilise cette méthode pour plusieurs raisons dont la réuséabilité, l'empreinte mémoire, la taille du binaire et les mises-à-jour facilités des différentes bibliothèques (par exemple, si une faille est découverte dans **printf**, nous n'aurons pas besoin de recompiler l'ensemble des programmes utilisant printf, il suffira simplement d'upgrader la bibliothèque)
- A contrario, dans la liaison statique, les différentes "définitions" des fonctions "externes" vont se retrouver intégrées au sein même de votre binaire (vous aurez donc par conséquent un binaire plus gros).

La quasi-totalité des logiciels utilisent la liaison dynamique.
Mais si nous détournions ces fonctions "externes" sans toucher à notre binaire d'origine ?
Il existe plusieurs méthodes pour effectuer un hook de ce style, voyons la plus simple pour l'instant :)

Mise en pratiqun pratique

Afin d'étudier cette méthode simplement, nous allons d'abord l'observer sur un programme d'exemple "cible" codé par nos soins, se voulant très simpliste, et dont voici le code source :

```
#include <openssl/conf.h>
#include <openssl/evp.h>

int main(void) {
    const unsigned char key[128] = "ThisIsMyMagicalAndHiddenKey!";
    const unsigned char iv[128] = {
        0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08,
        0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08,
        0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08,
        0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08,
    };
```

```
    EVP_CIPHER_CTX *ctx = EVP_CIPHER_CTX_new();
    EVP_EncryptInit(ctx, EVP_aes_256_cbc(), key, iv);

    // (...)

    return 0;
}
```

Pour conceptualiser plus rapidement les tenants et les aboutissants de cette méthode, nous avons développé un bout de programme effectuant des activités cryptographiques simples. Ce programme ne fait rien de bien concret, nous n'avons pas besoin d'aller plus loin pour l'instant dans notre programme, notre but étant de détourner la fonction EVP_EncryptInit qui est une fonction de base dans OpenSSL pour capturer la clef de chiffrement stockée dans la variable interne **key**.
Pour décrire rapidement le programme, ce dernier ne fait rien d'autre que d'initialiser le moteur cryptographique d'OpenSSL afin de lancer une procédure de chiffrement via l'algorithme AES-256-CBC.
L'algorithme AES-256-CBC a besoin (principalement) de deux paramètres :

- **Une clef de chiffrement**: elle va être utilisée (directement ou indirectement) pour chiffrer nos données. Elle sera stockée dans notre exemple dans la variable **key**
- **Une vecteur d'initialisation** : pour faire (très) simple, l'algorithme AES-CBC est un chiffrement par bloc, chaque bloc est chiffré en prenant en compte le résultat du chiffrement du précédent bloc. Cependant, quid du tout premier bloc ? Sur quel précédent bloc va t'il se baser vu qu'il n'en existe aucun ? C'est là que le vecteur d'initialisation rentre en action, il va nous servir comme "faux résultat d'un précédent bloc" et va nous servir pour le chiffrement du premier bloc. Ce vecteur d'initialisation, souvent surnommé IV, est stocké dans notre variable **iv**.

Nous allons maintenant compiler notre petit programme d'exemple :

```
$ gcc -Wall example.c -o example $(pkg-config –libs –cflags libcrypto libssl)
```

Par défaut, notre compilation va générer un programme utilisant la méthode des liens dynamiques.
Si nous démarrons notre programme, nous avons …

```
$ ./example
$ _
```

… rien. C'est parfaitement normal :-)
Notre programme fonctionne correctement, il a simplement initialisé son moteur cryptographie AES-256-CBC avant de s'arrêter sans rien faire (nous ne faisons aucune opération de chiffrement après)
Si nous analysons les liens dynamiques vers les différentes librairies utilisées :

 # méthode classique avec ldd

```
$ ldd ./example
 linux-vdso.so.1 (0x...)
 libcrypto.so.3 => /lib/x86_64-linux-gnu/libcrypto.so.3 (0x...)
 libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x...)
 /lib64/ld-linux-x86-64.so.2 (0x...)
```

 # méthode via variable env et ld-so

```
$ LD_TRACE_LOADED_OBJECTS=1 ./example
 linux-vdso.so.1 (0x...)
 libcrypto.so.3 => /lib/x86_64-linux-gnu/libcrypto.so.3 (0x...)
 libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x...)
 /lib64/ld-linux-x86-64.so.2 (0x...)
```

Avec les 2 ou 3 librairies classiques (libc par exemple), nous voyons que notre programme utilise une des parties de la librairie cryptographique OpenSSL via le module **libcrypto.so**
Pour simplifier : lors de son exécution, notre programme va piocher dans la libcrypto et utiliser deux fonctions qu'il ne possède pas en "interne" de son programme :

```
`EVP_CIPHER_CTX_new()`
`EVP_EncryptInit()`
```

Ces deux fonctions sont implémentées dans libcrypto, dont leurs codes sources se trouvent ici :

```
`EVP_CIPHER_CTX_new` : openssl/crypto/evp/evp_enc.c
`EVP_EncryptInit` : openssl/crypto/evp/evp_enc.c
```

Nous pouvons analyser l'ensemble des appels avec **ltrace** (la sortie a été nettoyée pour des raisons de visibilité) qui va nous lister les différents appels des fonctions vers les bibliothèques :

```
$ ltrace -ff ./example
EVP_CIPHER_CTX_new(1, 0x..., 0x..., 0x...)
EVP_aes_256_cbc(0, 0, 0, 0)
EVP_EncryptInit(0x..., 0x..., 0x..., 0x...)
+++ exited (status 0) +++
```

Nous constatons nos différents appels des fonctions OpenSSL implémentées dans notre programme, et leurs activités respectives.
Imaginons que nous ne connaissions pas le code source de cette application, mais nous voudrions extraire la clef secrète. Il existe plusieurs méthodes (`strings`, `objdump`, `gdb`, `radare2`, pour avoir quelques exemples), mais ici, nous allons essayer de se substituer à la fonction **EVP_EncryptInit**.
Pourquoi **EVP_EncryptInit** ? car c'est elle qui a besoin de la clef secrète pour initialiser le moteur cryptographique, dont voici sa définition :

```
int EVP_EncryptInit(EVP_CIPHER_CTX *ctx, const EVP_CIPHER *type,
        const unsigned char *key, const unsigned char *iv);
```

Avec cette information, démarrons notre implémentation.

Implémentation de notre hook

Nous utilisons le terme hook, mais voyez cela plutôt comme un Doppelganger.
Un doppelganger est une sorte de double maléfique :)
Nous allons créer le doppelganger de la fonction **EVP_EncryptInit**. Pour cela, nous allons créer un fichier que nous allons nommer **doppelganger.c** pour notre exemple et réimplementer exactement la fonction **EVP_EncryptInit** avec l'ensemble de ses arguments définis comme dans la documentation (ou son implémentation)

```
#include <stdio.h>
#include <openssl/evp.h>
```

```
int EVP_EncryptInit(EVP_CIPHER_CTX *ctx, const EVP_CIPHER *type,
        const unsigned char *key, const unsigned char *iv) {
  return 0;
}
```

Rien de plus... (pour l'instant)
Maintenant, nous allons vérifier si notre doppelganger est parfaitement accepté lors de la greffe.
Pour cela, nous devons effectuer deux actions :

- Compiler notre doppelganger sous forme de module (.so)
- Utiliser la méthode ld preload

Compilation de notre module doppelganger

Nous allons compiler notre programme en utilisant quelques paramètres spécifiques lors de la compilation car nous avons besoin qu'il soit sous forme de module :

```
$ gcc -fPIC "doppelganger.c" -shared -o "doppelganger.so"
```

Nous nous retrouvons avec un module .so que nous pouvons pré-analyser avec un petit file:

```
$ file "doppelganger.so"
doppelganger.so: ELF 64-bit LSB shared object, x86-64, version 1
(SYSV), dynamically linked, BuildID[sha1]=xxxxxxxx, not stripped
```

Nous constatons que nous avons bien un "Sshared object".
Voyons maintenant le chargement dynamique de cette nouvelle librairie au sein de notre application.

Chargement de notre module doppelganger

Un programme "dynamique" sous Linux est géré (entre autres) par le "dynamic link loader", aka ld.so.
Essayez d'exécuter votre ld.so :

```
$ /lib64/ld-linux-*so* —help
Usage: /lib64/ld-linux-x86-64.so.2 [OPTION]... EXECUTABLE-FILE [ARGS...]
You have invoked 'ld.so', the program interpreter for
dynamically-linked ELF programs.  Usually, the program interpreter is
invoked automatically when a dynamically-linked executable is started.
You may invoke the program interpreter program directly from the
command line to load and run an ELF executable file; this is like
executing that file itself, but always uses the program interpreter you
Invoked, instead of the program interpreter specified in the executable
file you run.  Invoking the program interpreter directly provides
access to additional diagnostics, and changing the dynamic linker
behavior without setting environment variables (which would be
inherited by subprocesses).
(...)
```

Oui, ld.so est un programme. Plus encore, c'est un interprétateur de binaire dynamique. Quand vous exécutez un binaire dynamique sous Linux, vous exécutez de facto ld.so ! (une sorte de wrapper si vous voulez)
Il existe deux manières d'injecter notre petit module :

- La méthode la moins connue en utilisant ld.so comme programme :

```
$ /lib64/ld-linux-x86-64.so.2 —preload "./doppelganger.so" "./example"
```

- La méthode la plus connue est simplement de définir la variable LD_PRELOAD :

LD_PRELOAD permet d'indiquer un module que nous souhaitons intégrer et charger lors du démarrage d'un programme. Elle va être interprétée par ld.so et changer le comportement lors du chargement des bibliothèques :

```
$ LD_PRELOAD="./doppelganger.so" "./example"
$ _
```

Aucune sortie comme auparavant. Au moins, notre application n'a pas planté :-)
Notez que ce comportement est normal dans notre exemple, nous verrons par la suite que notre doppelganger minimaliste de **EVP_EncryptInit** fera stopper ou planter les autres programmes. Analysons ce qu'il se passe avec ldd :

```
$ LD_PRELOAD="./doppelganger.so" ldd "./example"
 linux-vdso.so.1 (0x...)
 ./doppelganger.so (0x...)
 libcrypto.so.3 => /lib/x86_64-linux-gnu/libcrypto.so.3 (0x...)
 libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x...)
 /lib64/ld-linux-x86-64.so.2 (0x...)
```

Nous constatons que, maintenant, notre module "doppeldanger.so" est intégré dans la liste des bibliothèques chargées par le programme.
Si nous effectuons de nouveau un ltrace dessus :

```
$ LD_PRELOAD="./doppelganger.so" ltrace -ff "./example"
EVP_CIPHER_CTX_new(1, 0x..., 0x..., 0x...)
EVP_aes_256_cbc(0, 0, 0, 0)
EVP_EncryptInit(0x..., 0x..., 0x..., 0x...)
+++ exited (status 0) +++
```

En dehors des adresses, nous avons le même genre d'output qu'auparavant
Maintenant que nous constatons que notre greffe fonctionne, ce qui nous intéresse maintenant, sera de "manipuler" l'intérieur de notre **EVP_EncryptInit**.
Pour cela, reprenons le code de notre doppelganger en ajoutant simplement quelques lignes :

```
int EVP_EncryptInit(EVP_CIPHER_CTX *ctx, const EVP_CIPHER *type,
        const unsigned char *key, const unsigned char *iv) {
  if ( key != NULL ) {
     BIO_dump_fp(stdout, (const char *)key, 128);
  }
  return 0;
}
```

Nous n'avons ici ajouté que quelques lignes utiles :

- Un simple vérificateur de valeur (`if key != NULL`) afin d'éviter d'éventuels plantages sur une valeur nulle
- Une fonction interne à OpenSSL - un helper appelé **BIO_dump_fp** - qui va afficher le contenu de notre variable **key** (notez que nous aurions pu tout autant utiliser un simple **printf**, mais **BIO_dump_fp** ajoute un petit affichage sympathique :-) (notez que pour des raisons de visibilité dans l'article, cet affichage a été modifié)

Recompilons notre module et lançons notre programme dans la foulée :

```
$ gcc -fPIC "doppelganger.c" -shared -o "doppelganger.so"
$ LD_PRELOAD="./doppelganger.so" ./example
```

```
0x54 0x68 0x69 0x73 0x49 0x73 0x4d 0x79 0x4d 0x61 0x67 0x69 0x63 0x61 0x6c
0x41 0x6e 0x64 0x48 0x69 0x64 0x64 0x65 0x6e 0x4b 0x65 0x79 0x21
"ThisIsMyMagicalAndHiddenKey!"
```

Nous venons de détourner la fonction **EVP_EncryptInit** de notre applicatif et d'extraire notre clef secrète et cela, sans modifier notre programme d'origine !

Utilisation sur un programme externe

Maintenant, que nous avons testé la méthode sur un applicatif que nous maîtrisons, utilisons notre doppelganger directement sur un autre programme, par exemple **openssl** :

```
$ openssl aes-256-cbc
```

Avec ces arguments, OpenSSL va vous demander une clef de chiffrement pour débuter le chiffrement, puis va rester bloqué car il s'attend à obtenir des données en stdin, donc faites directement CTRL+C :

```
$ LD_PRELOAD="./doppelganger.so" openssl aes-256-cbc
enter AES-256-CBC encryption password:
Verifying - enter AES-256-CBC encryption password:
CTRL+C
$ _
```

Et... nous avons rien d'autre !... notre doppelganger ne marche pas ?

Nous allons maintenant étudier pourquoi. Pour cela, nous allons faire simple, avec notre ltrace des familles :

```
$ LD_PRELOAD="./doppelganger.so" ltrace -ff openssl aes-256-cbc
(...)
CRYPTO_malloc(512, 0x..., 630, 0)
CRYPTO_malloc(0x..., 0x..., 630, 0)
BIO_new_fp(0x..., 0, 2, 0)
EVP_CIPHER_get0_name(0x..., 0, 0x..., 0)
BIO_snprintf(0x..., 200, 0x..., 0x...)
EVP_read_pw_string(0x..., 512, 0x..., "enter AES-256-CBC encryption password:")
BIO_new_fp(0x..., 0, 2, 0)
RAND_bytes(0x..., 8, 0x..., 1)
BIO_write(0x..., 0x..., 8, 0)
BIO_write(0x..., 0x..., 8, 0)
EVP_BytesToKey(0x..., 0x..., 0x..., 0x...)
OPENSSL_cleanse(0x..., 512, 0, 0)
BIO_f_cipher(0x..., 0, 0, 0)
BIO_new(0x..., 0, 0, 0)
BIO_ctrl(0x..., 129, 0, 0x...)
EVP_CipherInit_ex(0x..., 0x..., 0, 0)
EVP_CipherInit_ex(0x..., 0, 0, 0x...)
BIO_push(0x..., 0x..., 0, 0)
BIO_ctrl(0x..., 10, 0, 0)
BIO_ctrl(0x..., 2, 0, 0)
BIO_read(0x..., 0x..., 8192, 0)
CTRL+C
$ _
```

Dans ce charabia très fortement réduit toujours pour des raisons de visibilité, vous aurez les mêmes trois contraintes : il faudra taper 2 fois le mot de passe et un CTRL+C sur **BIO_read()** car il s'attend à lire sur le stdin.

> ltrace peut être très verbeux, vous pouvez filtrer avec l'argument -e :

Exemple : ltrace -ff -e "OPENSSL*" -e "BIO*" -e "EVP*" openssl

Si on analyse les fonctions entre notre fonction **EVP_read_pw_string** (mais qui nous intéresse pas car elle ne fait que lire votre input lors de la saisie du mot de passe) et jusqu'à **EVP_CipherInit_ex**, nous voyons un petit **EVP_BytesToKey** entre :

```
BIO_new_fp(0x..., 0, 2, 0)
RAND_bytes(0x..., 8, 0x..., 1)
BIO_write(0x..., 0x..., 8, 0)
BIO_write(0x..., 0x..., 8, 0)
EVP_BytesToKey(0x..., 0x..., 0x..., 0x...)   <==== ici
OPENSSL_cleanse(0x..., 512, 0, 0)
BIO_f_cipher(0x..., 0, 0, 0)
BIO_new(0x..., 0, 0, 0)
BIO_ctrl(0x..., 129, 0, 0x...)
EVP_CipherInit_ex(0x..., 0x..., 0, 0)
```

Cette fonction est intéressante, elle a une tâche très précise à effectuer, celle de faire dériver notre clef à l'aide de certains paramètres.

Nous n'allons pas décrire le principe de la dérivation de clefs, elle est en dehors de notre scope, gardez juste en mémoire qu'une dérivation de clef va prendre en entrée notre clef d'origine et sortir une autre clef plus complexe (normalement…).

Cette dérivation va être utilisée par **EVP_CipherInit_ex**. Si nous effectuions un doppelganger de **EVP_CipherInit_ex**, nous n'aurions pas la clef de chiffrement, nous n'aurions que sa dérivation. Dans certains cas, cela peut avoir une utilité mais dans notre cas, cela complexifie l'étude, il faut donc taper plus haut pour obtenir l'endroit précis où la clef est manipulée, et **EVP_BytesToKey** est un bon candidat.

Voyons sa définition :

```
int EVP_BytesToKey(const EVP_CIPHER *type,const EVP_MD *md,
        const unsigned char *salt,
        const unsigned char *data, int datal, int count,
        unsigned char *key,unsigned char *iv);
```

Créons notre doppelganger :

```
int EVP_BytesToKey(const EVP_CIPHER *type, const EVP_MD *md,
        const unsigned char *salt,
        const unsigned char *data, int datal, int count,
        unsigned char *key, unsigned char *iv) {
  if ( data != NULL ) {
    BIO_dump_fp(stdout, (const char *)data, datal);
  }
  return 0;
}
```

Pourquoi utiliser **data** et non **key** ?

La documentation nous renseigne sur la nature de chaque :

❝ data is a buffer containing datal bytes which is used to derive the keying data.
The derived key and IV will be written to key and iv respectively. ❞

key et **iv** ne serviront que pour les outputs, notre input est stocké dans la variable **data**.

Recompilons et relançons :

```
$ gcc -fPIC "doppelganger.c" -shared -o "doppelganger.so"
$ LD_PRELOAD="./doppelganger" openssl aes-256-cbc
```

```
enter AES-256-CBC encryption password:
Verifying - enter AES-256-CBC encryption password:
0x4d 0x61 0x43 0x6c 0x65 0x66 0x55 0x6c 0x74 0x72 0x61 0x53 0x65 0x63
0x72 0x65 0x74 0x65 0x21  "MaClefUltraSecrete!"
(CTRL+C)
```

Notre clef a été interceptée !
Notez que si vous essayez un chiffrement avec notre doppelganger simpliste, votre cryptographie sera erronée : notre **EVP_BytesToKey** ne fera pas la dérivation, **key** et **iv** seront corrompus. Mais notre but étant d'intercepter simplement la clef pour l'instant :-)
Effectuons la même manipulation avec un autre paramètre OpenSSL en utilisant **pbkdf2** :

```
$ LD_PRELOAD=./doppelganger.so openssl aes-256-cbc -pbkdf2
enter AES-256-CBC encryption password:
Verifying - enter AES-256-CBC encryption password:
^C
```

Notre doppelganger ne (encore) marche plus ?
C'est parce que le paramètre **pbkdf2** demande à OpenSSL d'utiliser une autre méthode de dérivation de clef, donc nous ne passerons plus dans **EVP_BytesToKey** mais par une autre fonction de dérivation de clef.
Voyons cela de nouveau avec un ltrace :

```
EVP_read_pw_string(0x..., 512, 0x..., "enter AES-256-CBC encryption password:")
BIO_new_fp(0x..., 0, 2, 0)
RAND_bytes(0x..., 8, 0x..., 1)
BIO_write(0x..., 0x..., 8, 0)
EVP_CIPHER_get_key_length(0x..., 4, 0x..., 0)
EVP_CIPHER_get_iv_length(0x..., 4, 0x..., 0)
PKCS5_PBKDF2_HMAC(0x..., 4, 0x..., 8)        <=== ici
__memcpy_chk(0x..., 0x..., 32, 64)
__memcpy_chk(0x..., 0x..., 16, 16)
OPENSSL_cleanse(0x..., 512, 16, 16)
BIO_f_cipher(0x..., 0, 16, 16)
BIO_new(0x..., 0, 16, 16)
BIO_ctrl(0x..., 129, 0, 0x...)
EVP_CipherInit_ex(0x..., 0x..., 0, 0)
EVP_CipherInit_ex(0x..., 0, 0, 0x...)
BIO_push(0x..., 0x..., 0, 0)
BIO_ctrl(0x..., 10, 0, 0)
BIO_ctrl(0x..., 2, 0, 0)
BIO_read(0x..., 0x..., 8192, 0)
```

Effectivement, nous n'utilisons plus **EVP_BytesToKey**, nous utilisons maintenant la fonction **PKCS5_PBKDF2_HMAC** pour générer notre dérivation de clefs, dont voici la définition :

```
int PKCS5_PBKDF2_HMAC(const char *pass, int passlen,
        const unsigned char *salt, int saltlen, int iter,
        const EVP_MD *digest,
        int keylen, unsigned char *out);
```

Cette fonction ressemble de beaucoup à notre précédente, donc allons implémenter son doppelganger :

```
int PKCS5_PBKDF2_HMAC(const char *pass, int passlen,
            const unsigned char *salt, int saltlen, int iter,
            const EVP_MD *digest,
            int keylen, unsigned char *out) {
    if ( pass != NULL ) {
        BIO_dump_fp(stdout, (const char *)pass, passlen);
    }
    return 0;
}
```

Recompilons et exécutons de nouveau :

```
$ gcc -fPIC "doppelganger.c" -shared -o "doppelganger.so"
$ LD_PRELOAD="./doppelganger.so" openssl aes-256-cbc -pbkdf2
enter AES-256-CBC encryption password:
Verifying - enter AES-256-CBC encryption password:
0x4d 0x61 0x47 0x72 0x61 0x6e 0x64 0x65 0x43 0x6c 0x65 0x66 0x53 0x65
0x63 0x72 0x65 0x74 0x65 0x21  "MaGrandeClefSecrete!"
PKCS5_PBKDF2_HMAC failed
```

Et voilà, nous venons d'intercepter la clef de nouveau !
Notez que la dérivation plante car elle n'est pas conforme, le processus de chiffrement s'arrête, c'est normal, notre doppelganger ne fait rien de plus que d'extraire la clef et de ne rien retourner. Le programme s'arrête de lui-même.
Bien entendu, nous avons ici un exemple très simpliste, nous avons une interaction directe avec OpenSSL et donc nous connaissons la clef de départ (que nous donnons à openssl), mais nous pourrions continuer sur d'autres programmes utilisant d'autres méthodes cryptographiques ou différentes authentifications. Nous pourrions également détourner d'autres fonctions pour des buts totalement différents. Les perspectives sont (quasi) infinies…

Conclusion

Nous avons vu que nous pouvions détourner les appels des fonctions en interne d'un programme déjà établi et sans trop de contrainte avec peu d'outils pour étudier celui-ci.
Nos exemples se veulent simples. Mais nous pourrions imaginer avoir des doppelgangers à certains endroits - même sans besoin du LD_PRELOAD - au sein de notre système, effectuant certaines actions spécifiques à certains moments.
De plus, nos doppelgangers d'exemples, de par leur simplicité, peuvent se faire détecter rapidement car ils retournent de mauvaises données ou ont des comportements non prévus. Mais nous pourrions effectuer un véritable wrapper de la fonction d'origine depuis notre fonction doppelganger qui va effectuer correctement son travail - avec en complément - un code "malicieux". Vos programmes continueront de marcher normalement mais avec des activités annexes non voulues s'exécutant en arrière-plan.
Ici, dans nos exemples, nous ne faisons qu'un affichage brut d'une donnée, sans rien de plus. Mais imaginons avoir un code ouvrant une socket vers un serveur distant et poussant des données à chaque fois qu'une clef est demandée, générée ou utilisée, ou également un attaquant voulant laisser une porte dérobée sur un système qu'il a auparavant pénétré et voulant garder un accès permanent pour revenir plus tard et sans se faire repérer.
Nous pourrions faire cela avec n'importe quel programme, il ne faut donc jamais oublier ceci lors d'un développement : **même compilé, le comportement d'un programme est toujours étudiable, modifiable et donc détournable**.
Nous verrons dans un **prochain article** comment éviter ce genre de désagrément…

Implémenter une stratégie d'automatisation des tests

Interview menée par Marc Hage Chahine
(QA Practice Manager K-Lagan, Blog : la taverne du testeur)

Bonjour, Julien, peux-tu te présenter ?
Bonjour, je suis Automation Practice Manager chez K-LAGAN. Je travaille dans l'automatisation des tests depuis 2012, je suis l'un des co-créateurs du Jeu les 1001 Tests et du jeu Bugs End. J'anime des conférences et des webinaires autour de la qualité.

Pourquoi avoir choisi le métier d'expert automatisation ?
Lorsque j'ai découvert en 2011 l'automatisation, j'ai apprécié ce métier, car il possède une partie technique avec de la programmation d'automate, mais aussi une partie de réflexion et de méthodologie qualité. L'expertise m'a paru une suite naturelle, car je recherche le challenge et souhaite être au fait des dernières tendances.

Peux-tu décrire ce qu'est ce métier pour toi ?
Le métier d'expert automatisation est pour moi un métier technique (programmation, architecture…), d'écoute pour appréhender les contextes client, de transmission afin de pouvoir former les équipes ou informer/convaincre les projets autour de l'automatisation et d'apprentissage continu.

Quelle était ta première mission en automatisation ? Comment s'est-elle déroulée ?
J'ai commencé chez Altran avec une mission à l'Institut de Radioprotection et de Sureté Nucléaire (IRSN). J'étais chargé de l'automatisation (et référent QA) des tests sur 3 plateformes. Je suivais les TMA et étais le premier contact de l'infogérance.

Quelle est ta méthode pour implémenter l'automatisation ?
Je suis un processus en 4 étapes :

- **L'étude de faisabilité fonctionnelle et technique de l'automatisation.** L'objectif est d'étudier l'intérêt fonctionnel et financier de l'automatisation, mais aussi de commencer l'architecture de la solution d'automatisation.
- **La réalisation d'un POC.** l'objectif est de programmer un ou deux parcours représentatifs afin de valider la solution technique. Je rédige ensuite un rapport avec les informations de l'étude de faisabilité, les résultats du POC, les estimations de gains (ROI financier, temps d'exécution, le nombre de cas de tests exécutés et la charge libérée) avec nos conclusions.
- **La stratégie d'automatisation :** rédaction de la stratégie d'automatisation pour formaliser tous les processus et l'architecture de l'automatisation.
- **Généralisation :** réalisation, exécution et maintenance de l'automate. Mise en place d'un pilotage par les gains.

Quels sont les avantages de ta méthode ? Que fais-tu si une équipe refuse de l'adopter ?
Les avantages de cette technique sont de pouvoir arrêter l'intégration de l'automatisation au plus tôt si on constate une alerte de non-pérennité de l'automatisation dans le contexte client.

Quelles sont les difficultés les plus fréquentes lors de l'implémentation de l'automatisation ?
Les difficultés les plus fréquentes rencontrées lors de l'implémentation de l'automatisation sont un manque de maturité en qualité sur les projets (ex. : répertoires de test incomplets), environnement et JDD non maîtrisés et un développement qui ne respecte pas certaines bonnes pratiques (blocage de l'automate ou coût de maintenance trop élevé).
Aujourd'hui, les blocages liés à certains types de modules techniques (captcha, claviers numériques aléatoires…) sont automatisables avec de l'IA.

Comment es-tu devenu expert en automatisation ?
Je suis devenu expert en automatisation après être monté en compétence sur plusieurs outils et différents types d'applications et dans différents domaines.
L'auto-formation à de nouveaux outils et méthodologies, ma participation à des communautés internes pour partager mes connaissances m'ont permis d'être reconnu par les experts de mon entreprise.

Formes-tu des automaticiens ? Si oui, comment ?
Oui, j'ai formé des automaticiens.
Pour le comment, tout dépendait du niveau de la formation et des participants. J'axe principalement mes formations sur de la manipulation concrète (langage, automate, exécution…), car pour moi rien ne vaut la pratique. De plus, j'essaye de faire en sorte que mes exercices d'apprentissage soient ludiques et diversifiés (ex. : pour de l'apprentissage du code : réaliser un jeu simple comme le tic-tac-toe).

Quels profils recherches-tu ?
Je cherche surtout des personnes qui sont motivées avec au minimum un bagage technique ou une appétence pour la technique en ayant déjà une expérience dans le monde de la qualité logicielle.
Pour des collaborateurs plus expérimentés, les profils dépendront de la formation.

Quel est l'outil indispensable dans l'automatisation ?
Aujourd'hui, je n'ai pas forcément un outil indispensable, mais une liste dans lequel il est intéressant de connaître au moins un outil pour commencer :

- Pour de l'IHM : Robotframework, Selenium, Cucumber, Playwright ou Cypress
- Pour de l'API : Postman, ReadyAPI ou langage de programmation
- CICD : Jenkins ou GitLab CI

Quel est, selon toi, le prochain challenge des automaticiens ?
Pour moi, le prochain défi des automaticiens est l'arrivée de l'IA dans l'automatisation.
Je pense plutôt à une appropriation de l'IA dans l'automatisation à travers l'intégration aux outils ou une automatisation des recettes.

Embellir vos QR Codes à l'ère de l'IA générative

Raphaël Semeteys

Architecte IT et responsable des relations développeurs chez Worldline à Paris, possède plus de 25 ans d'expérience dans l'informatique. Spécialiste des logiciels libres et open source, il a animé un centre de compétences pendant neuf ans. Passionné de yoga depuis plus de 30 ans, il allie expertise technologique et bien-être holistique, offrant une vision enrichie de l'écosystème numérique. Pour en savoir plus : raphiki.github.io

À l'heure où l'intelligence artificielle transforme en profondeur le domaine de la création visuelle, les QR Codes connaissent eux aussi une nouvelle révolution. Autrefois simples outils utilitaires, ils deviennent désormais véritables œuvres graphiques grâce à l'intégration des technologies d'IA. Dans cet article, nous explorons comment allier personnalisation esthétique et robustesse technique pour concevoir des QR Codes uniques, attractifs et toujours fonctionnels.

QR Codes Traditionnels

Les QR Codes (« *Quick Response Codes* ») ont été développés en 1994 par Masahiro Hara et sont aujourd'hui reconnus comme une norme ISO/IEC. Ils représentent une évolution des codes-barres 2D, capables d'encoder des données numériques, alphanumériques, binaires ou en kanji sous la forme d'un motif de carrés noirs sur fond blanc. Ces codes existent en différentes tailles (ou versions), allant de la version 1 (21 x 21 carrés) à la version 40 (177 x 177 carrés).

De nombreuses bibliothèques et outils permettent de générer des QR Codes. Ma bibliothèque open source préférée est **QR Code Generator**, qui prend en charge toutes les fonctionnalités standard et existe en Java, TypeScript/JavaScript, Python, Rust, C++ et C. Par ailleurs, mon outil tout-en-un favori est **QR Toolkit**, une application Vue/Nuxt offrant la personnalisation des marqueurs et modules, ainsi que des mécanismes de vérification et de comparaison : une ressource précieuse pour ajuster les QR Codes. **Figure 1**

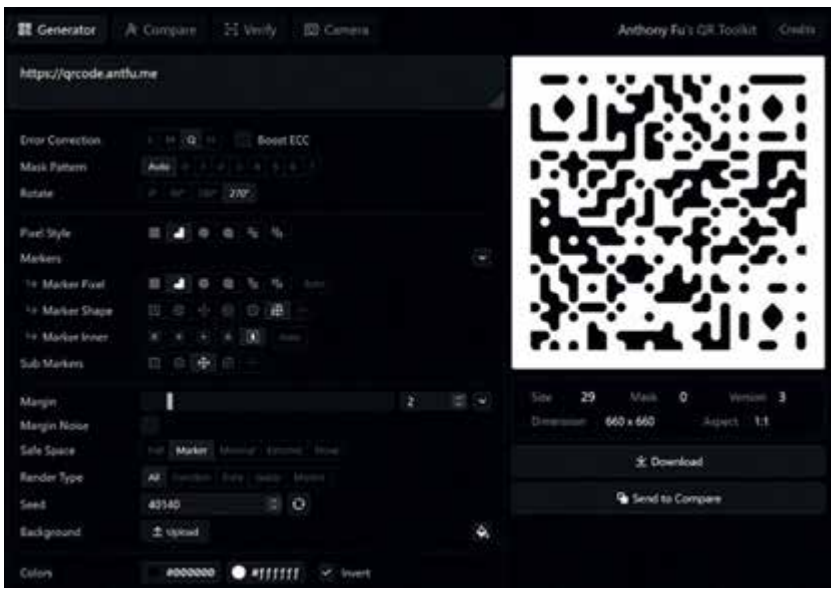
Les QR Codes comprennent plusieurs éléments essentiels pour garantir leur lisibilité par les scanners, dont trois marqueurs de position, des motifs d'alignement et de synchronisation, et un système de masquage. Sans entrer dans ces détails ici, j'aimerais plutôt me concentrer sur le mécanisme intégré de correction d'erreurs. Il utilise des codes de Reed-Solomon — aussi employés dans les supports de stockage (CD/DVD, RAID6) et les communications réseau (DSL, satellite) — en ajoutant des *codewords* supplémentaires dans la grille QR pour la correction d'erreurs. La norme définit quatre niveaux de correction, chacun associé à un pourcentage de tolérance différent.

Niveau	Tolérance d'erreur approximative
Low	~7%
Medium	~15%
Quartile	~25%
High	~30%

Cela signifie qu'un QR Code avec un niveau de correction élevé reste lisible même si jusqu'à 30% de l'image devient illisible. Cette fonction est souvent utilisée pour insérer une image au sein du QR Code : l'image intégrée est alors interprétée comme une zone d'erreur lors du scan.

Depuis des années, cette technique permet de personnaliser les QR Codes. Cet article explore une approche innovante de la personnalisation de QR Codes en s'appuyant, cette fois, sur l'IA générative.

Figure 1

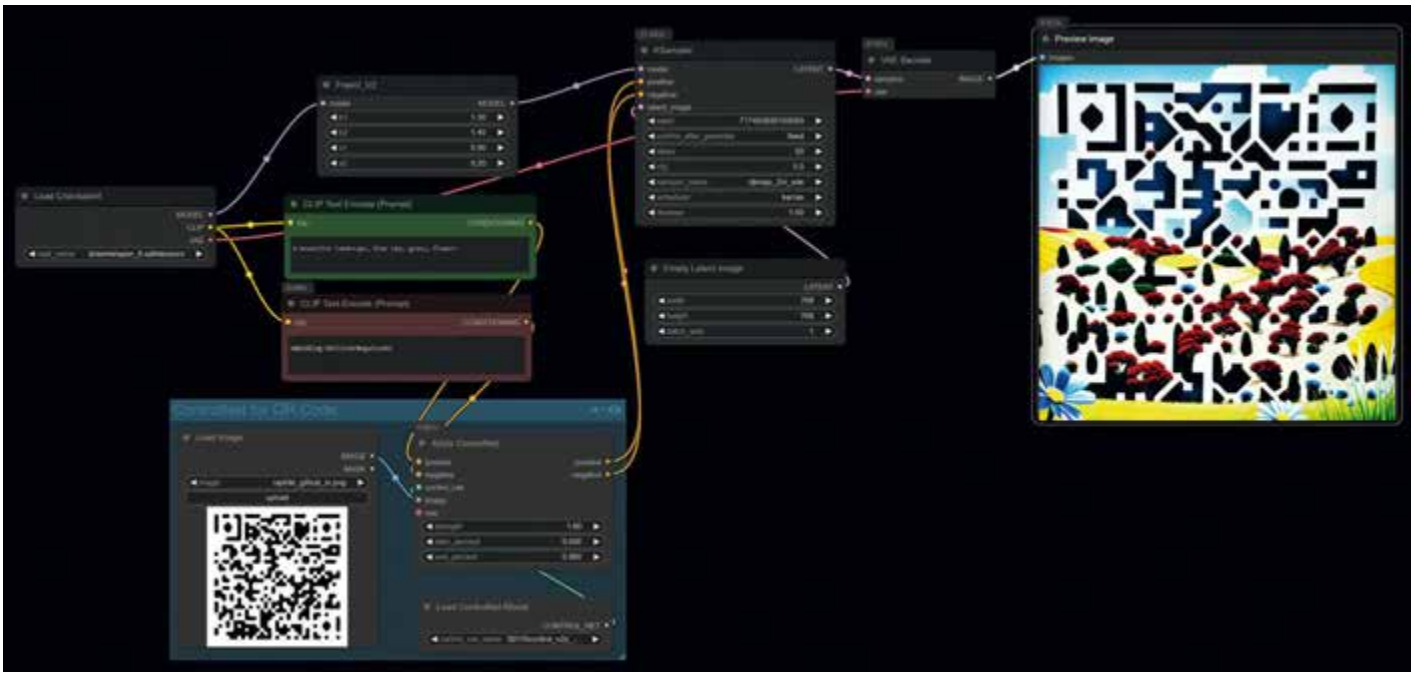


Exploiter l'IA Générative

Mon idée consiste à utiliser un modèle **Stable Diffusion** intégré dans l'interface graphique **ComfyUI** afin de concevoir et exécuter des workflows de génération d'images localement sur un PC équipé d'un GPU.

Pour modifier et affiner des QR Codes existants tout en préservant leur lisibilité, nous allons utiliser un ControlNet spécialisé nommé **QR Code Monster**. Les ControlNets sont des réseaux de neurones auxiliaires qui injectent des instructions ciblées dans le processus de génération en se concentrant sur des caractéristiques spécifiques de l'image d'entrée. Chaque ControlNet met l'accent sur certains aspects : structure (posture, contours, segmentation, profondeur), texture, disposi-

Figure 2



tion du contenu (boîtes englobantes, masques) ou style (cartes de couleurs, textures). Pour notre cas, il s'agira de préserver ou modifier les contrastes du QR code.
Procédons donc à la création d'un workflow ComfyUI combinant Stable Diffusion 1.5, le ControlNet QR Code Monster et un QR Code généré avec QR Toolkit. **Figure 2**

En ajustant les paramètres tels que la force du ControlNet, ses positions de départ et de fin, ainsi que le processus de sampling (ex. : 50 étapes), j'obtiens un résultat qui reste parfaitement lisible et qui correspond à mon prompt : « *Un beau paysage, ciel bleu, herbe, fleurs.* »

Cela montre comment Stable Diffusion, associé au ControlNet, a conservé la structure du QR Code tout en y injectant les éléments visuels souhaités. Grâce à la fonction de comparaison de QR Toolkit, il est possible d'évaluer la lisibilité du code en comparant les marqueurs de différence.
Figure 3

Ensuite, on peut modifier le prompt pour générer plusieurs variantes du QR code.

Figure 3

S'il est facile de changer le style général (premier exemple), il reste plus difficile d'intégrer directement un contenu précis dans le QR Code par rapport aux outils traditionnels (deuxième exemple). Pour approfondir cela, nous allons distinguer deux axes : le Style et le Contenu, avant de les combiner.

Personnaliser le style

L'enrichissement du prompt permet un contrôle plus fin sur l'aspect esthétique de votre QR code. Par exemple, on peut exploiter un LLM (*Large Language Model*) pour générer des descriptions détaillées telles que :
« *A pattern forged from molten lava, glowing with an intense fiery orange and red hue. Cracks in the surface reveal volcanic heat, with small embers rising around it.* »
(Traduction en français : un motif forgé dans la lave en fusion, brillant d'un orange et rouge incandescent. Des fissures révèlent la chaleur volcanique, tandis que de petites braises s'élèvent autour.)

De même, pour un style plus complexe et mystique :
« *An elegant, glowing elven door adorned with intricate, nature-inspired patterns and shimmering silver runes. Delicate vines and luminescent flowers intertwine with the carvings, pulsating with soft emerald and sapphire light. The archway, crafted from ethereal white stone, radiates a mystical aura, with faint golden mist swirling at its base, hinting at an ancient portal to a hidden realm* »

Figure 4

Figure 5

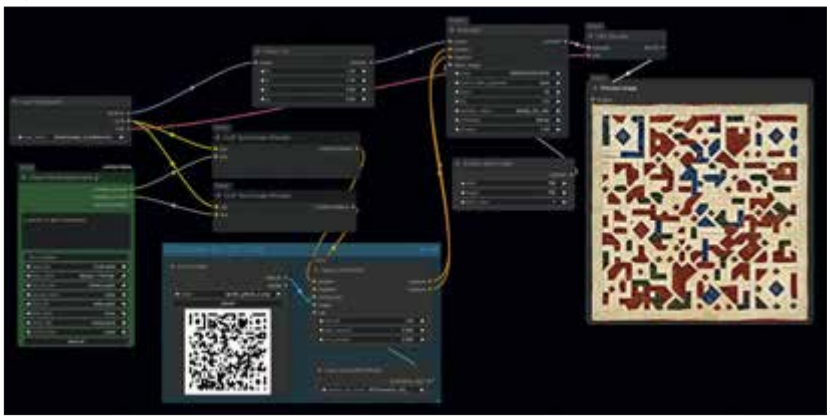


Figure 6

(Traduction en français: une porte elfique élégante et lumineuse, ornée de motifs inspirés de la nature et de runes argentées étincelantes. Des lianes délicates et des fleurs luminescentes s'entrelacent avec les gravures, palpitant d'une douce lumière émeraude et saphir. L'arche, taillée dans une pierre blanche éthérée, dégage une aura mystique, avec une fine brume dorée tourbillonnant à sa base, évoquant un ancien portail vers un royaume caché.)

Il est également possible d'injecter des styles prédéfinis dans les prompts à l'aide du nœud **iTools Prompt Styler Extra** dans ComfyUI : **Figure 4**

Ce nœud propose des prompts réutilisables, catégorisés selon différents styles artistiques : 3D, Art, Artisanat, Design, Dessin, Illustration, Peinture, Sculpture, Vectoriel, etc. L'intégrer dans le workflow facilite le test de styles variés sans modifier les autres paramètres. **Figure 5**

Voici quelques exemples de QR Codes créés avec des styles différents : **Figure 6**

On peut aussi combiner styles et prompts personnalisés pour obtenir des designs très personnels, et ainsi personnaliser sans limites l'apparence de vos QR codes.

Injecter du contenu

Maîtrisant la gestion du style, l'étape suivante consiste à incorporer un contenu généré spécifique dans le QR Code. Par exemple, je souhaite insérer l'image d'une posture de yoga. Vous pouvez consulter mes tutoriels (articles et vidéos) sur la génération d'images par IA pour comprendre le transfert de postures via les workflows.
Nous partons d'une image abstraite de la posture cible, ajoutons des ControlNets *Depth* et *Canny Edge* à notre workflow, et précisons dans le prompt : « *man, mixed race, short curly hair, black hair, 40 years old, white T-shirt, black yoga pants, short sleeves, smiling, viewing glasses, white background, barefoot.* » (traduction en français : homme, métis, cheveux courts frisés, cheveux noirs, 40 ans, t-shirt blanc, pantalon de yoga noir, manches courtes, souriant, lunettes, fond blanc, pieds nus.).
L'idée du prompt est de générer une image qui me ressemble. **Figure 7**

Pour obtenir une ressemblance réaliste, on ajoute aussi l'IP Adapter **FaceID** et le modèle post-traitement **FaceDetailer** au workflow. J'ai écrit un guide complet sur le transfert de visage, je ne le détaille pas ici. Le résultat reste lisible par un scanner et crée un QR Code intégrant la posture et l'identité souhaitées.

En utilisant QR Toolkit, la comparaison fait ressortir une vingtaine de nœuds différents, principalement autour du visage et du corps. **Figure 8**

Fusionner Style et Contenu

Tous les éléments sont désormais combinables, notamment en ajoutant le nœud iTools dans le workflow final :

Animer le QR Code

Puisque je peux intégrer un visage dans le QR Code, je peux également animer les expressions du visage avec des nœuds spécialisés. L'outil **Advanced Live Portrait** permet d'éditer, d'insérer et d'animer des expressions faciales dans les images. En insérant le QR Code généré, on peut animer mon visage pour faire apparaître un sourire ou un léger hochement de tête. **Figure 9**

L'animation produite peut être exportée en Animated GIF ou vidéo.

Conclusion

Ce court tutoriel a montré comment enrichir de façon spectaculaire l'aspect stylistique et le contenu d'un QR Code. Vous voilà prêt à créer des QR Codes engageants et personnalisés, en accord avec votre univers ou l'identité de votre marque.

Les seules limites sont votre patience et votre imagination, alors amusez-vous à expérimenter !

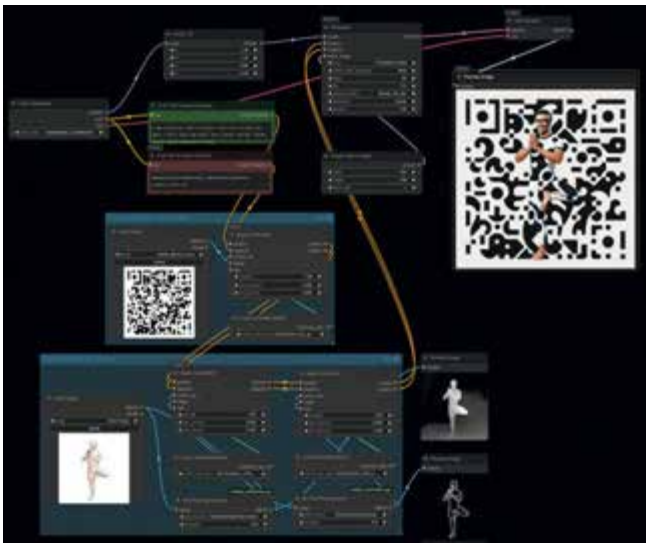


Figure 7

Figure 8

Figure 9

Haroun Azoulay
Ingénieur Logiciel / formateur et co-founder pour Beedigital

Python 3.13 et au-delà, que nous réserve le futur ?

Nous allons voir ensemble les deux évolutions majeures de Python 3.13. Nous aborderons le compilateur JIT ainsi que du Free-threaded sans GIL. Nous détaillerons dans un premier temps la partie pratique à travers du code expliqué. Ensuite, nous analyserons les résultats obtenus ainsi que les étapes de création et enfin, la théorie pure.

Il est important de noter que la majorité des améliorations de Python 3.13 resteront invisibles pour un utilisateur lambda. En effet, ce sont deux fonctionnalités expérimentales actuellement.

Utilisation et expérimentation du JIT (Just-In-Time Compiler) en Python

Mais déjà, qu'est-ce qu'un compilateur JIT ?

Un compilateur JIT (Just-In-Time) transforme pendant l'exécution du programme le code Python en code machine optimisé, ce qui permet d'accélérer l'exécution du script. Nous allons le voir dans la suite de l'article.

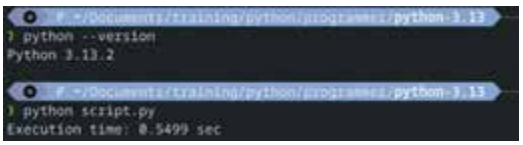
Mise en place d'une expérimentation simple

```
import time

def compute():
    total = 0
    for i in range(10**7):
    total += i
    return total

for _ in range(10):
    compute()

start = time.time()
compute()
end = time.time()

print(f"Execution time: {end - start:.4f} sec")
```

Dans cet extrait de code simple, je définis une fonction compute qui effectue une boucle for avec une incrémentation de 1, avec une condition d'arrêt de 10 puissance 7.
Une première boucle for _ in range(10) exécute cette fonction dix fois. Cette étape permet de "préparer" le code à une éventuelle optimisation.
Ensuite, la fonction est appelée une dernière fois, cette fois-ci chronométrée à l'aide de time.time(). Le temps d'exécution est affiché avec une précision de 4 décimales.

Sortie sans JIT

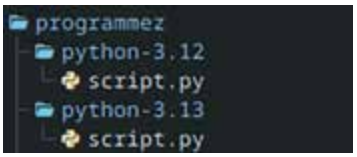
Sortie avec JIT



Mise en pratique

Structure du projet

Voici l'arborescence des dossiers :



Utilisation de pyenv

Au préalable, j'installe pyenv pour changer dynamiquement la version de Python. Je me rends sur le compte GitHub de celui-ci et je l'installe via le README (sur Docker, les résultats auraient été inexploitables, car on redémarre sur un environnement propre et cela empêcherait les optimisations adaptatives de JIT de s'exécuter).
Pour changer dynamiquement de version de Python sans perturber l'environnement global, pyenv a été utilisé en mode local (et non en global).
J'initialise les dossiers via les commandes ci-dessous (pour voir les versions déjà installées pyenv install —list)

Pour l'un :
pyenv local 3.12.10
Pour l'autre :
pyenv local 3.13.2

Une fois les versions de Python bien configurées, vous copiez le code écrit plus haut et l'exécutez.

Résultats observés

Le compilateur JIT permet généralement une amélioration des performances d'exécution de l'ordre de 5 à 15 %.
Dans l'exemple étudié, nous avons préparé l'optimisation JIT en appelant plusieurs fois la fonction compute avant de mesurer son exécution.
En accentuant volontairement l'effet de "chauffe", nous avons obtenu une réduction du temps d'exécution de 22,04 %, ce qui est loin d'être négligeable !
Ce résultat encourageant laisse entrevoir des améliorations encore plus significatives à l'avenir, notamment dans des contextes plus complexes ou des traitements répétitifs.
Nous reviendrons sur les hypothèses d'évolutions possibles à la fin de l'article.

Enjeux écologiques : Green IT

Au-delà des gains de productivité ou de performance brute, cette optimisation ouvre également la voie à une réflexion autour du Green IT. Réduire le temps de calcul, même de quelques %, signifie moins de cycles CPU (central process unit : c'est le processeur central) consommés, donc moins d'énergie utilisée.

Sur un langage aussi populaire que Python, présent dans des millions de scripts exécutés chaque jour, ces micro-améliorations peuvent avoir un impact réel à l'échelle globale.

En d'autres termes, l'introduction d'un mécanisme JIT contribue non seulement à accélérer les traitements, mais s'inscrit aussi dans une philosophie plus actuelle et enclin à notre ère : faire plus vite, avec moins de ressources.

Passage à la théorie : pourquoi parle-t-on d'un "compilateur" JIT ?

Le code Python est d'abord lu de haut en bas par le parseur de l'interpréteur. Lorsqu'une fonction comme compute() est déclarée, elle n'est pas exécutée immédiatement : elle est stockée en mémoire dans le heap (C'est une zone de la mémoire où sont stockés les objets créés dynamiquement pendant l'exécution d'un programme.) Sous forme d'un objet fonction prêt à être appelé plus tard. Ensuite, le code source est compilé en bytecode.

Le bytecode n'est pas du code machine, mais une représentation intermédiaire optimisée pour être exécutée rapidement par la machine virtuelle Python (PVM). Il se compose d'instructions simples (appelées opcodes).

Ensuite, le bytecode(opcodes) est interprété, instruction par instruction. Par la machine virtuelle Python (PVM), qui exécute les instructions en appelant des fonctions internes écrites en C.

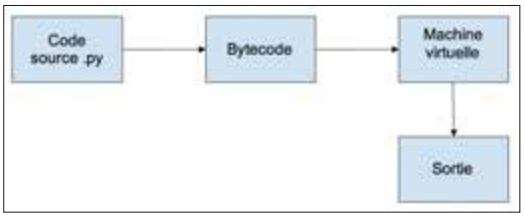
Nous entendons souvent parler de CPython, c'est le nom de l'implémentation officielle de l'interpréteur Python, qui regroupe tout : parseur, compilateur vers bytecode, PVM, gestion mémoire, etc.

Exemple :

```
def hello()
   print("Hello,Programmez!")



2     0 LOAD_GLOBAL      0 (print)
      2 LOAD_CONST       1 ('Hello, Programmez!')
      4 CALL_FUNCTION    1
```

Ensuite, place à la machine virtuelle Python qui traduira l'ensemble

Petit schéma résumant la situation :



Que fait le JIT exactement ?

Un compilateur JIT (Just-In-Time) est un composant qui, comme son nom l'indique, compile le code pendant son exécution. Contrairement à un compilateur classique qui transforme l'ensemble du code en langage machine avant de l'exécuter, le JIT agit en parallèle de l'interpréteur, au fur et à mesure des besoins.

On peut dire que le JIT surveille, en temps réel, l'activité de la machine virtuelle Python (PVM) : il observe quelles portions de code sont fréquemment utilisées, détecte les motifs d'exécution récurrents, et décide alors d'optimiser dynamiquement ces zones critiques.

Pour accélérer ce processus, ce compilateur JIT utilise une méthode appelée copy-and-patch :

L'idée est simple : plutôt que de générer tout le code machine à partir de zéro, le JIT copie des modèles de code machine prédéfinis (gabarits optimisés pour le processeur cible), puis les "patche" avec des données spécifiques comme les adresses mémoire des variables, les types ou les constantes utilisées.

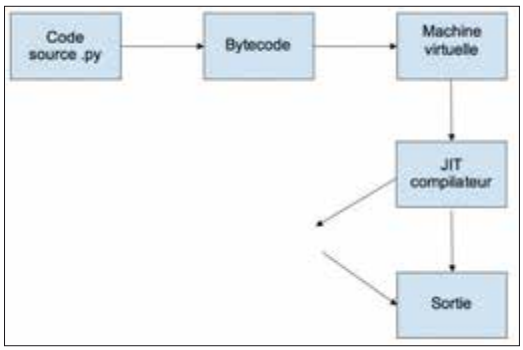
C'est un compromis efficace entre flexibilité et performance, permettant de générer du code rapide sans repartir de zéro à chaque compilation.

Quand Python détecte qu'une opération est toujours effectuée avec les mêmes types (par exemple : des entiers), il remplace le bytecode générique par une version spécifique et plus rapide adaptée à ces types.

Lorsque le JIT entre en jeu, il va au-delà du bytecode spécialisé :

Il génère du code machine optimisé (on parle de code assembleur) pour certaines portions du programme, qui peut être exécuté directement par le processeur sans passer systématiquement par l'interpréteur.

Et encore un schéma pour mieux comprendre :



Nous allons, maintenant, aborder un deuxième concept tout aussi fondamental pour comprendre l'évolution des performances de Python : le mode Free-threaded sans GIL.

Python sans GIL (Global Interpreter Lock) Introduction

Dans la continuité sur Python 3.13 après compilateur JIT, une nouvelle expérimentation a été réalisée autour d'une autre avancée majeure : l'exécution de Python sans GIL (Global Interpreter Lock).

Python 3.13 introduit en effet une version expérimentale dite "free-threaded", nous verrons le terme plus loin.

Comme précédemment, nous commencerons par observer les effets pratiques de cette nouveauté, avant d'en expliquer les fondements théoriques.

Mise en place d'une expérimentation simple

```
import time
from threading import Thread


COUNT = 50000000


def countdown(n):
    while n>0:
    n -= 1

t1 = Thread(target = countdown, args =(COUNT//4 ))
t2 = Thread(target = countdown, args =(COUNT//4 ))
t3 = Thread(target = countdown, args =(COUNT//4 ))
t4 = Thread(target = countdown, args =(COUNT//4 ))



start = time.time()
t1.start()
t2.start()
t3.start()
t4.start()
t1.join()
t2.join()
t3.join()
t4.join()
end = time.time()

print(f"Execution time: {end - start:.4f} sec")
```

Explication du fonctionnement

Dans cet extrait de code, nous importons le module threading. Mais un thread, c'est quoi ? Un thread est un fil d'exécution : c'est une séquence d'instructions que le système peut traiter de manière indépendante.

Plusieurs threads peuvent être lancés en parallèle dans un même processus, ce qui permet d'améliorer la fluidité et la rapidité des programmes.

L'exécution des threads est prise en charge par les cœurs physiques du processeur, qui se partagent la charge de travail.

Chaque thread (t1 à t4) exécute la fonction countdown, qui effectue une décrémentation simple à partir d'un nombre donné.

Ici, nous divisons la charge (COUNT) en quatre parts égales pour répartir le travail entre les quatre threads :

start() lance chaque thread.

join() permet au programme principal d'attendre la fin d'exécution de chaque thread avant de continuer.

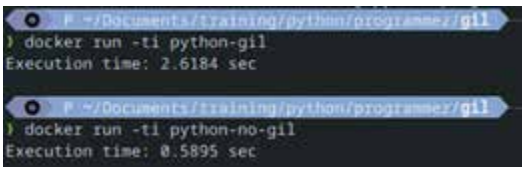
On mesure le temps total écoulé entre le lancement du premier thread et la fin du dernier.

Résultats observés

Sans GIL, le script s'exécute environ 4,5 fois plus rapidement que sur l'interpréteur. En effet, de manière triviale, le partage du travail à quatre est plus rapide que d'effectuer le travail seul. Ici, c'est exactement le même cas.

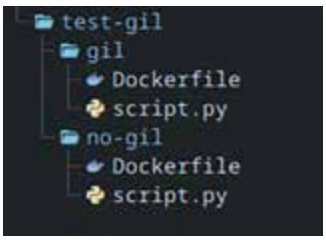
Cela s'explique par le fait que, sans GIL, les quatre threads peuvent également s'exécuter en parallèle, sur plusieurs cœurs du CPU.

Dans l'interpréteur standard avec GIL, un seul thread actif est autorisé à la fois pour exécuter du bytecode Python ce qui limite fortement les performances des traitements multi-thread CPU-bound, nous verrons pourquoi dans les prochains paragraphes.



Architecture du dossier

J'ai créé deux dossiers. Chaque dossier possède le même script et un Dockerfile (nous le voyons dans le prochain paragraphe).



Dockerfile

Contrairement au test précédent réalisé avec pyenv en local, ici, nous utilisons deux images Docker distinctes. Docker est un outil qui permet d'encapsuler une application et ses dépendances dans un conteneur. Ce conteneur garantit que l'application fonctionnera toujours de la même manière, quel que soit l'environnement sur lequel il est déployé. Cela simplifie la gestion, la portabilité et l'exécution rapide de tests, comme ici avec deux images différentes, l'une avec le GIL activé et la seconde désactivé. Pour l'une c'est une image de Python officiel classique tandis que pour l'autre image Docker, j'ai utilisé une image non officielle en testant au préalable que GIL est bien désactivé. Cette solution s'est avérée bien plus simple à mettre en œuvre pour les besoins de l'exemple.

Avec GIL

```
FROM python:3.13.3
WORKDIR /app

COPY . .

CMD ["python", "script.py"]
```

Sans GIL

```
FROM nogil/python
WORKDIR /app

COPY . .
```

```
CMD ["python", "script.py"]
```

Ensuite, à la racine de chaque dossier contenant un Dockerfile. Je crée l'image, à l'aide des commandes suivantes :

```
docker build -t "tag de votre choix"

Et enfin pour lancer l'image

docker run ""tag de votre choix"
```

Une fois l'expérimentation pratique faite, nous allons comprendre l'intérêt de GIL et son avenir au sein de Python.

Qu'est-ce que le GIL ?

GIL (Global Interpreter Lock) est un mécanisme qui limite volontairement l'exécution du bytecode à un seul thread à la fois. En conséquence, le processus d'un ou plusieurs thread se repose uniquement sur un thread à cause de GIL. On ne peut pas déverrouiller le multi thread, car l'interpréteur global (voir schéma plus haut de la machine virtuelle de Python) restreint l'ensemble en un seul.

En revanche, cette limitation ne s'applique pas aux opérations d'entrée/sortie (I/O) telles que lire un fichier, interroger un réseau ou attendre une base de données. Lors de ces opérations, le GIL est libéré temporairement, permettant à d'autres threads de s'exécuter pendant ce temps.

Le compteur de référence : l'intérêt de l'existence de GIL

Python possède se distingue des autres langages de programmation par la présence d'un compteur de référence. Avec l'aide du compteur de référence, nous avons le compte total de référence qui est fait en interne pour assigner une valeur à chaque objet de donnée.

Quand le compteur de référence est égal 0, l'objet est automatiquement libéré et donc il est ainsi libéré automatiquement via un cycle de nettoyage par le garbage collector.

Le garbage collector en Python est un mécanisme automatique qui sert à libérer la mémoire en supprimant les objets qui ne sont plus utilisés. Quand un objet n'est plus accessible par le programme, Python détecte qu'il est inutile et le libère pour récupérer la mémoire. Cela évite les fuites de mémoire et aide le programme à rester léger et performant.

Exemple d'un compteur de référence

```
import sys

x = [1, 2, 3]

print(sys.getrefcount(x))

y = x

print(sys.getrefcount(x))

del y

print(sys.getrefcount(x))
```

Sur le premier print, nous aurons **2** au compteur de références : une pour x et une temporaire ajoutée par getrefcount() lui-même.

Ensuite, nous affectons la variable x à la variable y. Il y a donc deux vraies références (x et y) plus la référence temporaire du print, soit **3** au total.

Puis, en supprimant y avec del y, nous revenons à **2** références : une pour x, une temporaire pour print.

Le compteur de références en Python doit être protégé, car plusieurs threads pourraient essayer simultanément d'augmenter ou de diminuer la valeur. Cela pourrait entraîner des erreurs graves, comme des fuites de mémoire ou des incohérences dans la gestion des objets pouvant aller jusqu'à provoquer ce qu'on appelle couramment un *segfault ou un deadlock*. Un segfault est une erreur de segmentation qui se produit lorsqu'un programme tente d'accéder à une zone mémoire qui ne lui appartient pas, soit parce qu'elle n'a pas été allouée, soit parce qu'elle est réservée à un autre usage. Un *deadlock* peut bloquer l'exécution du programme car si deux ou plusieurs threads attendent indéfiniment un verrou détenu par l'autre.

Afin d'éviter la mise en place de nombreux verrous complexes sur chaque structure de données, Python a introduit un verrou global unique : le Global Interpreter Lock (GIL).

Ce GIL garantit qu'un seul thread à la fois peut exécuter du bytecode Python, assurant ainsi la cohérence et la stabilité de la gestion mémoire.

En résumé, le GIL peut être comparé à un *mutex* global (mutual exclusion), un mécanisme utilisé en programmation pour empêcher plusieurs threads d'accéder simultanément à une même ressource.

Pourquoi le GIL existe-t-il encore ?

Parce que Python 2 a la même implémentation et si nous le changeons dans le Python 3, cela peut poser problème. Python dépend fortement du langage C, car la machine virtuelle Python et C lui-même dépendent de GIL. De plus, l'écosystème Python s'appuie massivement sur des extensions C, comme NumPy, Pandas ou encore TensorFlow, qui supposent que le GIL protège automatiquement l'accès aux structures de données partagées.

Pourquoi espérer un avenir sans GIL pour Python ?

L'existence du GIL constitue un obstacle majeur à l'exploitation optimale des processeurs multicœurs, ce qui limite de manière significative les performances en multithreading.

Dans des domaines tels que l'intelligence artificielle, l'absence de GIL permettrait une intégration plus rapide et plus efficace des données, en tirant pleinement parti du parallélisme offert par les architectures modernes.

De manière générale, pour le traitement de données volumineuses, lever la contrainte du GIL ouvrirait la voie à

des applications Python bien plus performantes et capables de gérer de lourdes charges de calcul de manière concurrente.
Ainsi s'achève l'exploration du GIL. Nous allons aborder maintenant l'avenir de Python.

Évolutions de Python, les hypothèses ?

Python, historiquement connu pour sa simplicité d'utilisation, mais aussi pour ses limitations de performance, est en train de connaître une phase de transformation importante. Deux éléments clés illustrent cette évolution : les travaux autour du GIL (Global Interpreter Lock) et le développement de nouvelles stratégies de compilation avec JIT (Just-In-Time).
Aujourd'hui, les évolutions récentes cherchent à supprimer ou limiter l'impact du GIL :

- Python 3.13 introduit des expérimentations autour du Free-threaded CPython, où l'interpréteur fonctionne sans GIL, mais au prix de quelques compromis sur la compatibilité.
- À moyen terme, nous pouvons envisager un Python sans GIL pour les nouveaux projets, permettant une vraie parallélisation des tâches CPU sans avoir besoin d'utiliser des processus séparés. Cela ouvrirait des performances bien supérieures pour les applications lourdes comme les serveurs web, les moteurs d'analyse de données ou les traitements vidéo lourds.

Cependant, la transition sera progressive : pour assurer la compatibilité avec l'écosystème existant (des millions de bibliothèques tierces), Python pourrait maintenir deux modes d'exécution : avec ou sans GIL, selon les besoins.
Parallèlement à l'évolution du GIL, un autre chantier majeur est en cours : l'intégration d'un compilateur JIT dans l'interpréteur standard de Python. Cela évite les multiples couches d'interprétation et accélère considérablement l'exécution.
Le plan, à long terme, est d'améliorer ce JIT de manière progressive, en visant un compromis entre gain de vitesse et consommation mémoire raisonnable.
À l'avenir, un Python avec JIT intégré pourrait devenir presque aussi performant que des langages compilés comme C ou Go pour certains types de programmes. Cela ouvrirait la voie à l'utilisation de Python dans des domaines où il était auparavant écarté pour des raisons de performance pure, comme :

- Des traitements d'image et vidéo en temps réel
- De la simulation scientifique complexe
- Et pourquoi pas ? Des moteurs de jeux

Il est également probable que plusieurs profils d'optimisation voient le jour : un Python "classique" pour les scripts rapides et un Python "optimisé" pour les applications gourmandes en performance.
Entre la suppression progressive du GIL et l'arrivée d'un JIT natif, Python est en train de réconcilier sa facilité d'usage historique avec les exigences modernes de performance.
Ces évolutions ne sont pas encore finalisées, mais elles indiquent clairement que Python s'adapte pour continuer à être un langage de premier choix, y compris pour les applications exigeantes.
Dans cette transition, nous nous devons de rester attentifs sur les PEP à venir (Python Enhancement Proposal). C'est un document officiel qui propose une amélioration du langage Python. Mais dans l'ensemble, l'avenir de Python semble plus prometteur que jamais, combinant accessibilité et puissance.

Nouveautés .NET 9 : focus sur Blazor et ASP.NET

Clément Sannier

Freelance DigitalCrafts. Architecte logiciel, ingénieur, développeur autour des technos Blazor , Dotnet et Azure.

Blazor a connu une transformation majeure ces dernières années, consolidant sa position comme un framework de développement web incontournable pour l'écosystème .NET et de Microsoft. Depuis son introduction en 2018, Blazor a évolué à travers différentes versions, passant d'une technologie émergente basée sur WebAssembly à une plateforme robuste intégrant des capacités serveur et client dans une seule architecture.

Avec l'arrivée de .NET 8, Blazor Full Stack a marqué un tournant en combinant les forces du rendu côté serveur (SSR) et du rendu côté client (CSR) dans un modèle unifié. Cette approche a permis aux développeurs de ne plus avoir à choisir dès le départ entre Blazor Server et Blazor WebAssembly, leur offrant ainsi la possibilité de basculer dynamiquement entre ces modes selon les besoins de l'application.

L'objectif de Microsoft avec .NET 9 est d'affiner encore cette intégration en améliorant les performances et la flexibilité de Blazor. Les nouvelles fonctionnalités visent à optimiser le mode Auto. Affiner la gestion du rendu en continu et renforcer l'interopérabilité avec JavaScript pour faciliter l'intégration avec des frameworks existants.

Blazor se positionne ainsi comme une alternative de plus en plus crédible face aux solutions JavaScript classiques, tout en restant profondément ancré dans l'écosystème .NET. Grâce à l'amélioration continue du framework, il devient plus simple de créer des applications web interactives sans sacrifier les performances ni la maintenabilité du code.

Blazor Full Stack et ses nouveautés en .NET 9

Avec .NET 9, Blazor Full Stack évolue pour offrir un développement encore plus fluide et performant. Le mode Auto est optimisé, permettant un basculement intelligent entre rendu serveur et client en fonction des interactions utilisateur. Microsoft améliore aussi la gestion du rendu hybride, réduisant la latence et optimisant le chargement progressif des composants. Enfin, l'intégration avec les API JavaScript et WebAssembly est renforcées pour une meilleure interopérabilité.

Rendu dynamique et améliorations des performances

Blazor dans .NET 9 introduit des optimisations significatives pour améliorer les performances et l'expérience utilisateur. L'une des évolutions majeures concerne le rendu en continu, qui permet d'afficher une page de manière progressive. Lorsqu'une page contient des traitements longs, comme des requêtes complexes en base de données ou des calculs intensifs, Blazor envoie un premier rendu rapide avec des espaces réservés avant d'injecter les données une fois prêtes.

Le mécanisme repose sur l'attribut @rendermode, qui contrôle le mode de rendu du composant. Avec l'option InteractiveAuto, le rendu commence côté serveur pour un affichage immédiat, puis bascule en WebAssembly pour améliorer l'interactivité.

Dans cet exemple, la page commence avec un message temporaire indiquant le chargement des données. Blazor effectue un premier rendu minimaliste, puis récupère les données en arrière-plan avant d'actualiser l'interface utilisateur.

Blazor 9 améliore aussi la gestion du cycle de vie des composants en réduisant les allers-retours inutiles entre le client et le serveur. L'introduction du rendu différé (Streaming Rendering) permet d'afficher les parties statiques d'une page immédiatement, puis d'ajouter dynamiquement les contenus dépendants de traitements asynchrones. **Figure 2**

Avec StreamRendering, Blazor affiche immédiatement les éléments disponibles et injecte les autres composants dès que leurs données sont chargées, améliorant ainsi la fluidité perçue.

Ces améliorations renforcent les performances de Blazor, notamment sur les applications nécessitant des mises à jour dynamiques fréquentes, rendant le framework plus compétitif face aux solutions JavaScript modernes.

Nouveaux composants et améliorations des API

Blazor dans .NET 9 introduit plusieurs nouveaux composants et améliore les API existantes pour offrir plus de flexibilité aux développeurs. Parmi les évolutions notables, l'intégration de nouveaux contrôles natifs et l'amélioration de la gestion des événements enrichissent l'expérience utilisateur.

Une des nouveautés majeures concerne la gestion des formulaires et des entrées utilisateur. Le composant InputFile permet désormais un téléchargement optimisé des fichiers avec la possibilité de traiter les données en flux, réduisant ainsi la consommation mémoire. **Figure 3**

```
@page "/weather"
@rendermode InteractiveAuto

<h1>Météo</h1>

@if (forecasts == null)
{
    <p>Chargement des données...</p>
}
else
{
    <ul>
        @foreach (var forecast in forecasts)
        {
            <li>@forecast.Date.ToShortDateString() - @forecast.TemperatureC°C</li>
        }
    </ul>
}
```

Figure 1

```
@page "/weather"
@attribute [StreamRendering]
```

Figure 2

Cet exemple illustre le traitement de fichiers en streaming, permettant d'envoyer des fichiers volumineux sans les charger entièrement en mémoire.

Blazor .NET 9 apporte aussi des optimisations sur le DOM virtuel et l'arbre de rendu. L'introduction du RenderTreeDiff Builder permet une gestion plus fine des modifications dans l'interface utilisateur, réduisant le nombre d'éléments recalculés à chaque interaction.

Ces améliorations renforcent l'efficacité du framework et facilitent le développement d'applications interactives tout en réduisant la consommation de ressources.

Interopérabilité avec JavaScript

Blazor dans .NET 9 améliore son interopérabilité avec JavaScript, facilitant l'intégration avec des bibliothèques et frameworks externes. L'une des avancées majeures est la gestion des objets JavaScript persistants, qui permet de réduire la latence en évitant des appels répétés aux scripts. Désormais, il est possible de charger un module JavaScript une seule fois et de l'appeler dynamiquement sans nécessiter de nouvelle requête, optimisant ainsi les performances des applications. L'appel aux API JavaScript bénéficie aussi d'une gestion optimisée des paramètres et des retours asynchrones. Cela simplifie la communication entre Blazor et des frameworks comme React ou Vue, tout en permettant une meilleure sérialisation des données échangées. Les interactions entre le client et le serveur sont également améliorées grâce à une synchronisation plus efficace, garantissant une réactivité accrue. L'une des avancées majeures concerne les objets JavaScript persistants. Plutôt que d'invoquer une fonction JS à chaque interaction, il est désormais possible de conserver un objet en mémoire côté Blazor et de l'appeler à la demande, réduisant ainsi la latence et améliorant les performances. **Figure 4**

Et voici le fichier JavaScript counter.js correspondant : **Figure 5**

Avec cette approche, Blazor charge le module JavaScript une seule fois et réutilise l'objet, évitant ainsi des appels répétitifs au moteur JS. Cette méthode offre une communication plus fluide avec les librairies JavaScript existantes, facilitant l'intégration avec des Frameworks comme React ou Vue.

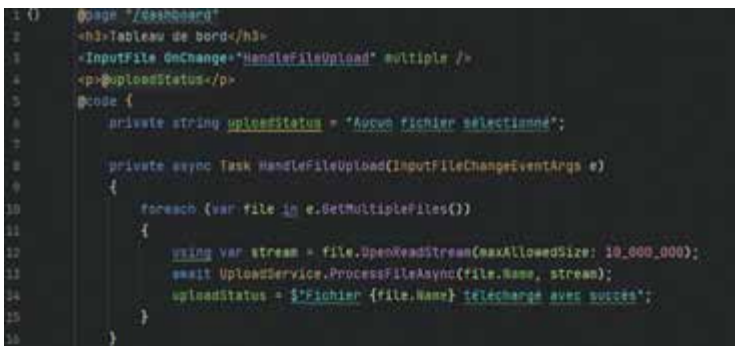


Figure 3

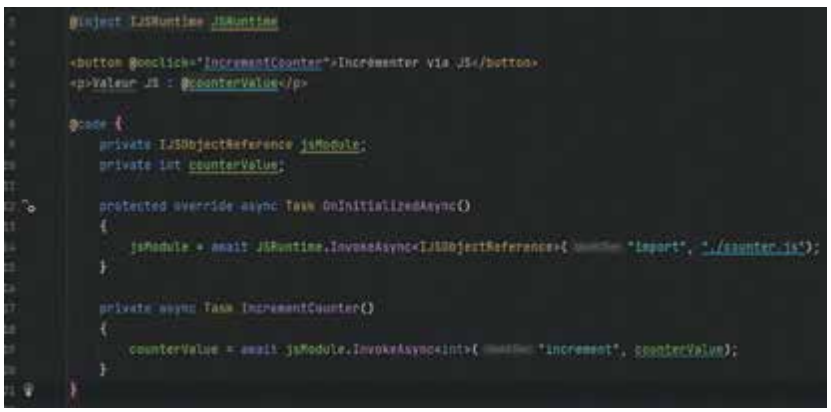


Figure 4

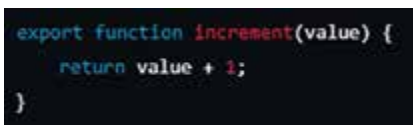


Figure 5

Amélioration du modèle hybride

Le modèle hybride de Blazor continue d'évoluer avec un équilibrage plus intelligent entre l'exécution côté client et serveur. Grâce au mode Auto, les composants peuvent initialement s'exécuter sur le serveur pour offrir un rendu rapide, puis basculer en WebAssembly lorsque les ressources sont disponibles, garantissant une meilleure fluidité des interactions.

Avec ces optimisations, Blazor .NET 9 se positionne comme une alternative robuste aux Framework JavaScript, offrant une combinaison unique de performances et de flexibilité pour le développement d'applications web modernes.

Nouveaux templates et facilitation du développement

Blazor dans .NET 9 simplifie la création d'applications grâce à de nouveaux templates optimisés. Ces modèles unifient les approches Blazor Server et Blazor WebAssembly, permettant aux développeurs de ne plus choisir entre les modes dès le départ. L'intégration du mode Auto par défaut facilite la gestion des rendus dynamiques sans nécessiter de configurations complexes.

Les templates offrent également une meilleure organisation du code avec une séparation plus claire entre les composants interactifs et les pages statiques. Cette approche permet une maintenance plus efficace et réduit le couplage entre la logique métier et l'interface utilisateur. De plus, les applications générées avec ces nouveaux modèles bénéficient directement des optimisations de performance, comme le chargement différé des composants et l'exécution hybride.

L'amélioration des outils de développement accompagne ces évolutions. L'intégration avec Visual Studio et les extensions CLI permettent une génération plus rapide des projets et une configuration simplifiée des dépendances. Les nouveaux templates introduisent également une meilleure gestion de l'interopérabilité avec JavaScript, facilitant l'intégration de bibliothèques tierces sans manipulation complexe du code.

Avec ces changements, Blazor devient encore plus accessible et performant, réduisant la barrière d'entrée pour les nouveaux projets tout en optimisant la productivité des développeurs expérimentés.

Injection de dépendances par constructeur dans Blazor

Avec .NET 9, l'injection de dépendances reste un pilier du développement Blazor, et Microsoft continue d'améliorer la gestion des services. Un service peut être injecté directement dans un composant via le constructeur : **Figure 6**

Ici, IUserService est injecté dans le constructeur du composant pour être utilisé dès l'initialisation.

Différence entre injection par constructeur et injection via [Inject]

L'injection par constructeur permet d'imposer explicitement la

dépendance et favorise le **test unitaire** plus facilement. En revanche, Blazor supporte aussi l'injection via l'attribut [Inject], souvent utilisé pour des services optionnels. Bien que pratique, cette approche ne permet pas d'initialiser les services dans le constructeur, ce qui peut être une limitation pour certaines architectures.

Améliorations en .NET 9

Avec .NET 9, Microsoft optimise la gestion des services injectés dans Blazor, notamment en réduisant la latence et en améliorant la gestion du cycle de vie des services singleton, scoped et transient. Cela permet une meilleure intégration avec les architectures modulaires et microservices, notamment grâce à un meilleur support du DI dans les nouveaux templates Blazor Full Stack.

Amélioration de l'expérience de reconnexion côté serveur

Blazor Server repose sur une connexion SignalR entre le client et le serveur. En cas de perte de connexion (changement de réseau, mise en veille, coupure internet), Blazor doit gérer la reconnexion sans perte d'état. Avec .NET 9, plusieurs améliorations rendent cette reconnexion plus fluide et robuste.

Tout d'abord, la reconnexion progressive permet d'essayer de restaurer la connexion sans recharger entièrement la page. Blazor effectue plusieurs tentatives silencieuses en arrière-plan avant d'afficher une alerte à l'utilisateur.

Ensuite, la persistance de l'état des composants a été optimisée. Lorsqu'une reconnexion réussit, l'application reprend exactement là où elle s'est arrêtée, sans nécessiter un cycle de rendu complet. Cela évite la perte des saisies utilisateur ou des interactions en cours. Blazor offre également une meilleure personnalisation de la reconnexion. Les développeurs peuvent désormais intercepter les événements de connexion et personnaliser l'expérience utilisateur. Voici un exemple pour détecter et afficher l'état de connexion : **Figure 7**

Ce code utilise CircuitHandler pour détecter les changements d'état de connexion et mettre à jour l'interface en conséquence. Enfin, il est possible d'ajuster le comportement de reconnexion en interceptant SignalR dans blazor.server.js : **Figure 8**

Ces améliorations rendent Blazor Server plus résilient, en particulier pour les applications métiers critiques et les environnements avec une connectivité instable.

Nouveauté ASP.Net Core

ASP.NET Core continue d'évoluer avec .NET 9, consolidant sa position de framework web performant et polyvalent. Microsoft met l'accent sur l'optimisation des performances, la simplification du développement et une interopérabilité accrue avec les architectures modernes.

L'un des axes majeurs de cette nouvelle version est l'amélioration du pipeline HTTP, réduisant la latence des requêtes et optimisant l'exécution des middlewares. Kestrel, le serveur web intégré, bénéficie de nouvelles optimisations pour gérer plus efficacement les connexions et améliorer la scalabilité des applications. D'autre part, les Minimal APIs continuent de s'enrichir avec un support amélioré pour le typage fort et la validation, rendant le développement backend plus fluide. La sécurité est aussi renforcée avec des mises à jour sur l'authentification OpenID Connect et OAuth.

Enfin, la prise en charge des architectures distribuées et des microservices s'améliore, avec un meilleur support de gRPC,

```
public partial class ConstructorInjection(IUserService userService) : ComponentBase
{
    private string userName;

    protected override async Task OnInitializedAsync()
    {
        userName = await userService.GetUserNameAsync();
    }
}
```

Figure 6

```
@if (!isConnected)
{
    <p>Connexion perdue. Tentative de reconnexion...</p>
}

@code {
    private bool isConnected = true;

    protected override void OnInitialized()
    {
        ConnectionTracker.ConnectionStateChanged += OnConnectionStateChanged;
    }

    private void OnConnectionStateChanged(bool connected)
    {
        isConnected = connected;
        StateHasChanged();
    }
}
```

Figure 7

```
window.Blazor.defaultReconnectionHandler = {
    onConnectionDown: () => console.log("Connexion perdue. Tentative de reconnexion..."),
    onConnectionRestored: () => console.log("Reconnexion réussie !"),
};
```

Figure 8

SignalR et Kubernetes. Ces évolutions font d'ASP.NET Core une solution encore plus robuste pour le développement d'applications web modernes et cloud-native.

Optimisation des performances et du pipeline HTTP

ASP.NET Core dans .NET 9 améliore encore les performances du traitement des requêtes grâce à des optimisations du pipeline HTTP. L'objectif est de réduire la latence et d'optimiser l'utilisation des ressources, notamment pour les applications à forte charge.

Kestrel, le serveur web intégré, bénéficie d'améliorations notables, avec une meilleure gestion des connexions concurrentes et une réduction du coût des allocations mémoire. Ces optimisations permettent un traitement plus rapide des requêtes et une meilleure évolutivité. Les améliorations du support HTTP/3 garantissent également une latence réduite et une gestion plus efficace du multiplexage des requêtes.

Le pipeline de requêtes a été optimisé en réduisant le nombre de middlewares nécessaires pour certaines opérations courantes. Cela permet d'accélérer l'exécution des requêtes tout en réduisant la consommation CPU. De plus, les améliorations de la gestion des tampons de requêtes et réponses permettent d'éviter des copies inutiles en mémoire, ce qui se traduit par un gain de performances notable.

Enfin, ASP.NET Core introduit de nouveaux outils de diagnostic et de profiling, facilitant l'analyse des performances des applications en production. Grâce à ces évolutions, les applications ASP.NET Core peuvent traiter davantage de requêtes avec une latence réduite et une consommation optimisée des ressources serveur.

Sécurité et authentification améliorées

ASP.NET Core dans .NET 9 renforce la sécurité des applications en améliorant les mécanismes d'authentification et d'autorisation. L'un des axes majeurs concerne l'intégration plus fluide avec OpenID Connect et OAuth, permettant une meilleure ges-

```
app.MapPost("/users", async (UserDto user, UserService service) =>
{
    var result = await service.CreateUserAsync(user);
    return Results.Created($"/users/{result.Id}", result);
})
.WithName("CreateUser")
.WithValidator<UserDto>();
```

Figure 9

```
app.MapGet("/users/{id:int}", async (int id, UserService service) =>
{
    var user = await service.GetUserByIdAsync(id);
    return user is not null ? Results.Ok(user) : Results.NotFound();
});
```

Figure 10

tion des sessions et des jetons d'accès. La prise en charge des flux d'authentification modernes a été optimisée pour sécuriser les API et les applications web tout en réduisant la complexité de configuration. L'authentification basée sur les certificats et les clés publiques bénéficie d'une meilleure gestion au sein du framework, avec des options avancées pour le renouvellement automatique des certificats et la validation des clés. Ces évolutions simplifient l'implémentation d'authentifications fortes, notamment pour les environnements nécessitant une haute sécurité comme les applications d'entreprise.

ASP.NET Core améliore également son système d'autorisation avec une gestion plus fine des stratégies et des rôles. Les nouvelles API permettent de centraliser la gestion des règles d'accès et d'appliquer des politiques plus complexes, sans impacter les performances.

Enfin, la protection des données et la gestion des sessions ont été optimisées avec un chiffrement renforcé et une meilleure gestion des jetons d'accès. Ces améliorations garantissent une sécurité accrue tout en maintenant de bonnes performances pour les applications nécessitant une gestion avancée des utilisateurs et des permissions.

Minimal API et évolutions du développement backend

ASP.NET Core dans .NET 9 continue d'améliorer les Minimal APIs, rendant le développement backend plus rapide et plus fluide. Ces évolutions visent à simplifier la création d'API en réduisant le code nécessaire tout en améliorant la lisibilité et la maintenabilité.

L'une des principales améliorations concerne le typage fort des paramètres et des réponses. Désormais, les Minimal APIs prennent en charge les validations automatiques, réduisant la nécessité d'écrire du code de validation manuel. **Figure 9**

Grâce à .WithValidator<T>(), ASP.NET Core applique directement les règles de validation définies pour UserDto, éliminant le besoin d'une validation manuelle dans le contrôleur.

L'intégration avec Entity Framework Core et Dapper a également été optimisée, facilitant l'accès aux bases de données. L'exécution des requêtes bénéficie d'une meilleure gestion des transactions et des connexions, permettant un traitement plus efficace des données. **Figure 10**

Les réponses des Minimal APIs bénéficient aussi de nouvelles optimisations, notamment une sérialisation JSON plus rapide et un encodage plus efficace des données. Les nouveaux filtres permettent de modifier automatiquement les réponses avant envoi, facilitant la transformation des données.

Enfin, la gestion des erreurs a été améliorée avec un meilleur support des statuts HTTP et une intégration directe avec les outils de diagnostic. Ces évolutions font des Minimal APIs un choix encore plus pertinent pour le développement rapide d'API performantes, tout en réduisant la complexité du code backend.

Interopérabilité avec les autres frameworks et microservices

ASP.NET Core dans .NET 9 améliore l'interopérabilité avec les architectures distribuées, facilitant l'intégration avec d'autres frameworks et microservices. Ces améliorations portent sur la communication entre services, la compatibilité avec les standards modernes et l'optimisation des performances pour les échanges de données.

Le support de gRPC a été renforcé, avec une gestion améliorée du streaming et une réduction de la latence des échanges entre services. Grâce à ces optimisations, les applications peuvent désormais traiter des flux de données en temps réel de manière plus efficace, réduisant l'empreinte réseau et améliorant la scalabilité. SignalR, utilisé pour la communication en temps réel, bénéficie également d'améliorations, notamment avec une meilleure gestion des connexions et une optimisation des performances sur les réseaux instables. La prise en charge du WebTransport, une alternative plus performante à WebSockets, permet de réduire la latence des mises à jour en temps réel et d'améliorer l'expérience utilisateur.

L'intégration avec Kubernetes et les environnements cloud a été simplifiée, notamment grâce à un meilleur support des sidecars et des stratégies de communication optimisées. Les applications ASP.NET Core peuvent maintenant interagir plus efficacement avec des services tournant dans des environnements conteneurisés, grâce à une configuration plus fluide des connexions réseau et de la découverte de services.

Enfin, la compatibilité avec les API RESTful a été étendue, permettant une meilleure gestion des contrats d'API et une interopérabilité facilitée avec des services écrits en d'autres langages. De nouveaux outils permettent de générer automatiquement des clients pour des API distantes, facilitant la consommation de services tiers au sein d'applications ASP.NET Core. Ces évolutions rendent ASP.NET Core encore plus adapté aux architectures modernes, en garantissant des échanges de données rapides et sécurisés entre services.

Déploiement et hébergement optimisés

ASP.NET Core dans .NET 9 améliore le déploiement avec un démarrage plus rapide des applications et une meilleure gestion des ressources. L'exécution en conteneurs est optimisée avec des images plus légères et un scaling automatique sur Kubernetes. Le support des environnements serverless, comme Azure Functions, réduit la latence des requêtes à froid. Enfin, la surveillance avec OpenTelemetry facilite le suivi des performances et des erreurs, rendant le déploiement plus efficace et flexible.

Conclusion

ASP.NET Core et Blazor dans .NET 9 apportent des améliorations majeures en performances, flexibilité et interopérabilité. Blazor Full Stack simplifie le développement web avec un rendu hybride optimisé, tandis qu'ASP.NET Core renforce la sécurité, les Minimal APIs et l'intégration cloud. Grâce aux optimisations du pipeline HTTP, des microservices et du déploiement, .NET 9 se positionne comme une plateforme moderne, performante et adaptée aux applications web évolutives.

Industrialiser l'éco-conception web avec méthode et efficacité

Mathieu COTTARD
Fondateur Greenoco
Adresse site :
https://greenoco.io
Adresse plateforme :
https://app.greenoco.io

L'éco-conception numérique n'est plus une option marginale. Pour les développeurs et intégrateurs, elle s'impose désormais comme une dimension incontournable d'un projet web moderne, aux côtés des performances et de l'accessibilité. Si les bonnes pratiques sont bien connues de la communauté tech, leur mise en œuvre reste souvent artisanale. C'est sur ce point que Greenoco entend changer la donne, avec une plateforme qui automatise l'audit et priorise les optimisations à fort impact, en quelques minutes.

Une approche pragmatique et industrialisée

Créée pour aider les équipes techniques à intégrer concrètement l'éco-conception dans leurs projets, Greenoco mise sur une approche systématique et scalable. L'objectif : permettre à tout développeur ou intégrateur de disposer rapidement d'un audit structuré et actionnable, sans avoir à lancer manuellement une batterie d'outils ou à interpréter des dizaines de métriques dispersées.

Dès son lancement, la plateforme s'est distinguée par sa volonté de rendre les audits d'éco-conception web aussi automatisables que les audits de performance. À la clé, un gain de temps significatif, et surtout une vision consolidée de l'impact environnemental réel d'un site, basée sur des données tangibles.

Un audit multi-pages, pondéré selon l'impact carbone

Contrairement à bon nombre d'analyses qui se concentrent sur la page d'accueil ou sur une vue trop limitée du site tel un parcours utilisateur, Greenoco propose un audit multipage, qui s'adapte à la structure réelle du site et à ses usages. Les pages sont pondérées en fonction de leur trafic (audience), ce qui permet d'évaluer l'impact carbone réel, page par page, à l'échelle du domaine.

Autre particularité : l'évaluation carbone ne repose pas uniquement sur les poids de fichiers ou le nombre de requêtes. Elle prend en compte le pays d'hébergement du datacenter, et si l'information est accessible, le PUE (Power Usage Effectiveness) du centre de données. En croisant ces éléments, Greenoco propose une estimation affinée de l'empreinte carbone du site, en grammes de CO équivalent, page par page.

Chaque optimisation potentielle est ensuite classée en fonction du gain carbone estimé, toutes audiences et pages confondues, ce qui permet aux équipes de prioriser leurs interventions de manière rationnelle : compression d'images, suppression de scripts inutiles, minification, mise en cache, etc.

Un cercle vertueux : performance, SEO et sobriété

Il est souvent inutile d'opposer performance web et éco-conception : dans une majorité de cas, les optimisations recommandées par Greenoco améliorent aussi les temps de chargement, et donc le SEO comme le Quality Score en SEA. Le gain environnemental devient un effet de levier supplémentaire pour des pratiques qui relèvent déjà du bon sens technique.

Vivatech 2025 - Greenoco

Côté praticité, l'audit complet ne prend que quelques minutes. L'interface propose une visualisation synthétique des pages analysées, avec un tri par empreinte carbone, par type de contenu lourd ou par typologie d'optimisation. Le tout pensé pour que l'équipe technique puisse passer à l'action immédiatement.

Malin, efficace, rapide

Greenoco coche trois cases rarement réunies dans le domaine de l'éco-conception numérique : efficacité, rapidité et simplicité d'intégration dans les workflows existants. Sans réinventer la roue, la plateforme permet de structurer, prioriser et justifier des optimisations techniques avec un cadre mesurable et actionnable.

C'est cette capacité à industrialiser la démarche, sans l'appauvrir, qui pourrait bien faire de Greenoco un compagnon

incontournable pour les développeurs soucieux d'allier qualité, performance… et sobriété.

Un badge d'éco-score lisible, automatisé et valorisant

Pour rendre visible l'engagement environnemental d'un site, Greenoco propose un **badge d'éco-score** à intégrer très simplement via une **iframe**. Ce badge repose sur un **score unique**, calculé à partir de l'audit multipage pondéré selon l'audience réelle du site. Contrairement à d'autres approches où le score change d'une page à l'autre, ici, la cohérence est assurée : le visiteur voit un seul indicateur, stable, lisible et représentatif de l'ensemble du site. Le badge renvoie vers un **certificat en ligne** reprenant des données détaillées : empreinte carbone globale du site, moyenne par page, dates d'audit, et surtout **réduction d'impact constatée depuis le premier audit**. Cette transparence facilite la communication des efforts accomplis, aussi bien en interne qu'auprès du public. Une fois mis en place, **aucune action supplémentaire n'est requise** : le badge comme le certificat sont mis à jour automatiquement après chaque nouvel audit. Plusieurs formats sont disponibles (dark/light, avec ou sans la note), pour s'adapter aux chartes graphiques. C'est le choix qu'a fait **elle-et-vire.com**, qui a réduit de **54 % l'empreinte carbone** de son site en suivant **12 recommandations prioritaires** ciblant les **10 pages les plus visitées**. Un exemple concret d'optimisation efficace, visible et valorisable, sans complexité technique.

Concrètement, une intégration instantanée, sans intrusion ni friction technique

Greenoco a été conçu pour s'intégrer sans aucune modification du site à auditer : **aucun script, tag ou code à installer**, ni sur le front, ni côté serveur. Tout se fait **depuis l'extérieur**, directement sur la plateforme app.greenoco.io. L'utilisateur renseigne un domaine, sélectionne les URLs à analyser (manuellement, via fichier CSV ou sitemap), puis indique l'audience de chaque page, soit manuellement, soit via un **connecteur Google Analytics sécurisé**. D'autres intégrations (Matomo, Piwik Pro…) sont en cours de développement. L'audit est lancé en quelques clics, sans configuration serveur, sans besoin DevOps, et les résultats sont disponibles rapidement. Pour valoriser les résultats, un **badge d'éco-score** s'intègre via une simple balise iframe (non intrusive), automatiquement mise à jour à chaque nouvel audit. Ce fonctionnement transparent, rapide et sécurisé permet une adoption immédiate dans les phases de diagnostic, d'optimisation ou de refonte, sans jamais toucher à l'infrastructure du site.

Des résultats exploitables immédiatement, avec des optimisations classées par impact

Une fois les données remontées et l'audit terminé (généralement en moins de 10 minutes), Greenoco restitue une analyse structurée et immédiatement exploitable. Chaque page analysée est accompagnée d'un diagnostic précis : poids total, nombre de requêtes, principaux types de ressources (images, JS, CSS, polices...), et surtout **préconisations d'optimisation concrètes**, directement issues de l'analyse. L'interface classe tous les **assets identifiés** par leur potentiel d'économie d'équivalent CO_2, en croisant leur poids, leur fréquence d'utilisation sur le site et leur impact carbone estimé. Ainsi, un fichier CSS commun à toutes les pages, compressable de quelques Ko, peut être considéré comme plus prioritaire qu'une grosse image isolée. Les optimisations proposées couvrent les **fichiers CSS, HTML et JS** (minification, compression), les **images** (redimensionnement à la taille affichée, conversion en .webp), et les **polices** (passage de .ttf ou .woff à .woff2). Chaque recommandation est accompagnée d'un gain potentiel mesuré, et peut être directement intégrée dans le backlog d'un sprint technique. L'interface permet aussi de **filtrer les résultats**, d'exporter les données, et de conserver l'**historique des audits**, utile pour suivre les progrès dans le temps ou justifier les efforts réalisés en interne ou auprès des clients.

Une roadmap technique ambitieuse, toujours dans une démarche d'éco-conception

Côté roadmap, les prochaines étapes incluent l'intégration de données d'audit d'accessibilité, (sur la base du WCAG et des retours Lighthouse, complété d'une matrice d'aide à la complétion de l'audit de conformité) ainsi que l'analyse avancée des **Core Web Vitals**, pour rapprocher encore davantage sobriété, UX et performance.

L'objectif : fournir un cockpit complet aux développeurs, où chaque action d'optimisation est guidée par son impact à la fois écologique, technique et business.

De futurs développements d'API et de version selfhost permettront également de répondre aux attentes des grands groupes, des plateformes SaaS d'audit de site, et des développeurs. L'API rendra également possible le développement de plugins pour CMS tels que Wordpress, Drupal ou Joomla.

Une démarche méthodologique rigoureuse, en collaboration avec le Pôle Éco-conception

Derrière la simplicité d'utilisation de Greenoco, il y a une volonté forte d'ancrer l'outil dans une **démarche scientifique robuste**. En collaboration avec le **Pôle Éco-conception**, soutenu par l'**ADEME**, une étude est actuellement en cours pour tester et affiner le modèle de calcul de l'empreinte carbone utilisé par la plateforme. L'objectif est double : **valider les hypothèses actuelles** (facteurs d'émission, prise en compte des infrastructures, pondération selon l'audience…) et **amé-**

UNE PLATEFORME GRATUITE POUR TESTER JUSQU'À 20 PAGES

Pour permettre aux développeurs de tester l'outil sans contrainte, Greenoco propose une version gratuite de sa plateforme — app.greenoco.io — qui permet d'auditer jusqu'à 20 pages d'un même domaine simultanément. De quoi évaluer rapidement les principaux leviers d'optimisation et découvrir le fonctionnement de l'outil.

À l'issue de l'audit, un **badge d'éco-score global** peut être généré et affiché sur le site analysé. Ce badge, déjà visible sur des sites comme elle-et-vire.com, permet de valoriser l'engagement environnemental du site tout en sensibilisant les visiteurs.

liorer la précision du modèle dans des contextes variés. Ce partenariat vise aussi à aller plus loin, avec le développement d'une **analyse multicritère** intégrant, en plus du carbone, **la consommation en eau** liée à la production et à l'usage du numérique. À plus long terme, Greenoco travaille à un **modèle d'Analyse du Cycle de Vie (ACV) simplifié**, qui permettra d'estimer aussi **l'impact des équipements physiques** impliqués dans le parcours utilisateur : matériel des datacenters selon leur usage réel, mais aussi **prorata de consommation des terminaux des utilisateurs finaux**, en fonction de leur temps de chargement et de navigation. Une vision systémique, nécessaire pour donner une image fidèle de l'impact environnemental global d'un site web.

Une plateforme éco-conçue jusque dans son code

Chez Greenoco, l'éco-conception ne s'applique pas seulement aux sites audités : elle commence par la plateforme elle-même. Tout le SaaS a été développé sur mesure, sans framework surdimensionné ni dépendances superflues, en intégrant dès le départ les bonnes pratiques de sobriété numérique : chargement différé des ressources, réduction du JavaScript, usage raisonné des librairies externes, ou encore compression et cache bien maîtrisés. Le front a été optimisé pour offrir une interface fluide tout en restant légère et rapide, même sur des connexions limitées. Côté back, la logique est la même : chaque audit ne conserve que les données nécessaires, les historiques non utiles sont automatiquement purgés entre deux analyses, évitant l'accumulation de volumes inutiles. L'infrastructure est également choisie avec soin, selon les besoins réels en puissance, et l'hébergement est assuré en France, dans des datacenters sélectionnés pour leur efficacité énergétique. Une démarche cohérente, qui s'inscrit dans une volonté de limiter l'impact environnemental global du service, y compris dans ses fondations techniques.

Greenoco vs ÉcoIndex : deux outils, deux approches

Bien connu des développeurs sensibles à la sobriété numérique, l'outil ÉcoIndex propose un indicateur simple basé sur trois paramètres techniques (nombre de requêtes, poids total de la page et complexité du DOM), ramenés à une échelle de A à G. Si cet indice a le mérite de la simplicité, il présente certaines limites lorsqu'il s'agit d'évaluer l'empreinte carbone réelle d'un site à l'échelle. Greenoco se positionne comme une alternative complémentaire, avec une méthodologie plus fine et opérationnelle. Contrairement à ÉcoIndex qui évalue une page à un instant T sans notion d'usage, Greenoco réalise des audits multi-pages, en tenant compte de l'audience réelle de chaque URL. L'analyse est donc pondérée selon la fréquentation, ce qui reflète bien mieux l'impact global du site. Autre différence clé : Greenoco ne se limite pas à un score, mais fournit des recommandations concrètes, priorisées par gain carbone potentiel, permettant une mise en œuvre directe par les équipes. De plus, l'audit est historisé, ce qui permet de suivre les progrès d'un site dans le temps. Enfin, plutôt que d'attribuer un score indépendant à chaque page, la plateforme génère un **éco-score consolidé** calculé dynamiquement à partir de la moyenne pondérée des pages auditées. Un choix méthodologique assumé, qui favorise une lecture globale et responsabilisante de l'impact numérique du site.

UNE SOLUTION FRANÇAISE, NÉE D'UN BESOIN CONCRET

Greenoco est une solution française, développée au Havre, en Normandie. Elle est née d'un constat simple : malgré la montée en puissance du GreenIT, aucun outil ne permettait d'industrialiser efficacement les audits d'éco-conception web dans des contextes projets réels. Face à ce manque, **Mathieu Cottard**, fondateur, a décidé de créer l'outil qu'il aurait aimé avoir sous la main pour ses propres sites. Rapidement rejoint par **Hugo Bollaert**, développeur fullstack devenu lead dev de la plateforme, ils décident de construire Greenoco entièrement **from scratch**, avec une architecture simple, robuste, et surtout **sans recours à l'intelligence artificielle**. Leur choix : automatiser des tests précis, transparents et compréhensibles, plutôt que de masquer les analyses derrière des modèles opaques. Depuis, la plateforme s'est étoffée tout en conservant cette approche pragmatique et directe, pensée pour les équipes techniques. Greenoco revendique pleinement son ancrage local et son indépendance technologique, et s'adresse aussi bien aux freelances qu'aux grands groupes, avec une vision claire : rendre l'éco-conception web accessible, actionnable, et mesurable à grande échelle, sans alourdir les processus de développement.

Benoît Prieur

Ingénieur logiciel et auteur aux Éditions ENI. En juin 2025, est sorti son dernier ouvrage à propos du Framework PyQt : Développez vos interfaces graphiques en Python avec PyQt6.

https://www.benoit-prieur.fr/

Sobriété énergétique : Python, C, Rust… tous les langages ne se valent pas

Alors que les enjeux écologiques invitent toutes les industries à réfléchir à leur impact environnemental, le monde du développement logiciel reste encore en retrait sur ces questions. Pourtant, les lignes de code que nous écrivons chaque jour ne sont pas neutres : elles consomment de l'énergie, de la mémoire, du temps machine. Et tous les langages ne se valent pas sur ces points.

Une étude scientifique rigoureuse publiée en 2017 (*Energy Efficiency across Programming Languages*, conférence SLE 2017), et actualisée jusqu'en 2021, a proposé une évaluation comparative des langages de programmation selon trois critères objectifs : la consommation énergétique, le temps d'exécution, et l'usage mémoire. Cette étude, **produite par une équipe de chercheurs portugais de l'université du Minho,** constitue à ce jour l'une des rares tentatives systématiques et reproductibles de mesure de l'empreinte des langages de programmation. Elle fournit ainsi une base scientifique utile pour repenser certains choix techniques à l'aune de la performance environnementale.

L'étude repose sur le *Computer Language Benchmarks Game* (CLBG), une initiative communautaire qui regroupe un ensemble de problèmes algorithmiques classiques (factorielles, tri, calculs d'arbre, etc.), chacun résolu dans de nombreux langages par des implémentations optimisées.

Les langages étudiés sont au nombre de 27 allant des grands classiques (C, C++, Java, Python, Ruby, Perl…) à des langages plus récents ou spécialisés (Rust, Go, Haskell…). Chaque langage est exécuté à travers une implémentation validée, réputée idiomatique et performante. L'objet de l'exécution est un jeu de 13 problèmes retenus pour lesquels chaque langage dispose d'une implémentation optimisée. Les problèmes sont, pour en citer quelques-uns : *fannkuchredux* (accès indexé à une petite séquence d'entiers), *mandelbrot* (génération d'un fichier bitmap portable représentant l'ensemble de Mandelbrot), *regex-redux* (recherche de motifs de 8-mers dans de l'ADN et substitution de motifs spéciaux), *binary-trees* (allocation, parcours et libération de nombreux arbres binaires), ou encore *thread-ring* (passage d'un jeton d'un thread à un autre).
Les langages étudiés sont classés dans le tableau ci-dessous par paradigme de programmation (impératif, fonctionnel, orienté objet, script), les langages multi-paradigmes (comme Rust par exemple) apparaissent donc plusieurs fois.

Paradigme	Langages
Fonctionnel	Erlang, F#, Haskell, Lisp, OCaml, Perl, Racket, Ruby, Rust
Impératif	Ada, C, C++, F#, Fortran, Go, OCaml, Pascal, Rust
Orienté objet	Ada, C++, C#, Chapel, Dart, F#, Java, Javascript, OCaml, Perl, PHP, Python, Racket, Rust, Smalltalk, Swift, TypeScript
Script	Dart, Hack, JavaScript, JRuby, Lua, Perl, PHP, Python, Ruby, TypeScript

L'ensemble du benchmark, pour chaque implémentation de chacun des 27 langages relatifs à chacun des 13 problèmes, tourne sur une machine unique sous Linux Ubuntu Server 16.10, avec comme matériel un Intel Core i5-4460, 4 cœurs, cadencés à 3,20 GHz, 16 Go de RAM. Toute mesure est effectuée sur une seule et même machine, sans tâche concurrente, dans un environnement contrôlé.

Trois mesures sont réalisées durant le benchmark pour chaque couple (problème, langage) : le temps d'exécution en millisecondes, la consommation système mesurée *via* des fonctions système, et la consommation énergétique, mesurée en joule via les interfaces Intel RAPL (*Running Average Power Limit*), intégrées au CPU, qui permettent une mesure fine de la consommation d'énergie par processus. Pour rappel, le joule est l'unité de base de l'énergie dans le système international ; il correspond à l'énergie nécessaire pour produire une puissance d'un watt pendant une seconde. À titre de comparaison, 1 kilowattheure (kWh) équivaut à 3 600 000 joules, soit l'énergie consommée par un appareil de 1 000 watts fonctionnant pendant une heure. Pour donner un ordre de grandeur, une recherche Google peut consommer jusqu'à 1 000 joules (données 2009), mais selon certaines analyses, cette valeur pourrait descendre en 2025 autour de 150 joules.

Des écarts énergétiques de 1 à 75

Les résultats, pondérés sur l'ensemble des problèmes testés, révèlent des écarts spectaculaires. Le langage C, le plus performant en la matière, est pris comme base unitaire des résultats. Ainsi parmi les langages très utilisés dans l'industrie du logiciel, C# est trois fois plus énergivore que le C, JavaScript environ 5 fois plus, et le langage Python, 75 fois plus énergivore que le C.
Ci-dessous le tableau des résultats consolidés sur les 13 problèmes exécutés pour chaque langage, pour la consommation énergétique, avec comme référence à 1, le langage le plus performant, le langage C. Sont indiqués les 5 langages les plus performants en la matière et les 5 langages les moins performants.

Langage	Énergie consommée
C	1
Rust	1.03
C++	1.34
Ada	1.70
Java	1.98
…	…
Lua	45.98
JRuby	46.54
Ruby	69.91
Python	75.88
Perl	79.58

On s'aperçoit que les langages compilés se retrouvent en tête de classement et les langages de

script sont plutôt mal classés, le meilleur élève dans cette catégorie étant le langage Dart (3,83 plus énergivore que le C).

La publication expose les résultats pour chaque problème selon le triplet de résultats (énergie, vitesse, mémoire), ce qui nous permet d'explorer plus en détail un problème en particulier et de chercher à établir des équivalences énergétiques pour ce problème. Prenons par exemple le problème *binary-trees* (allocation, parcours et libération de nombreux arbres binaires). La consommation en langage C est de 39 joules, là où la consommation en langage Python est d'environ 1800 joules. À l'échelle industrielle, par exemple un micro-service susceptible d'être appelé plusieurs centaines de milliers de fois chaque jour, disons un million d'appels, le différentiel C/Python s'élève à 1,76 gigajoules par jour, soit environ 486 kWh/jour. À titre illustratif, cela représente la consommation électrique journalière d'une cinquantaine de foyers français.

Trois critères croisés : énergie, vitesse, mémoire

L'une des forces de l'étude est de croiser les trois dimensions (consommation énergétique, vitesse d'exécution, usage de la mémoire), ce qui permet de mieux comprendre les ressorts de la consommation. Nous pouvons ainsi établir quelques observations générales.

- **Mémoire et énergie sont fortement corrélées :** les langages qui allouent peu de mémoire tendent à être plus sobres. C, C++ et Rust, qui laissent au programmeur une grande maîtrise de l'allocation mémoire, sont les plus efficaces.
- **Temps d'exécution ≠ énergie consommée :** un programme plus rapide n'est pas toujours plus sobre. Des langages rapides (comme Java) peuvent avoir une empreinte énergétique plus élevée que d'autres, plus lents, mais économes en mémoire.
- **La gestion des structures mutables/immutables joue aussi un rôle :** en Python, de nombreuses structures (*tuple*, *set*, *string*) sont immutables, ce qui nécessite des copies mémoires fréquentes. À l'inverse, en Rust ou en C, la gestion de la mémoire, non-déléguée à un ramasse-miettes, permet des optimisations fines et une économie drastique de ressources.

Faut-il alors réécrire tout notre backend en C ou en Rust ? Évidemment non. Le choix d'un langage de programmation dépend d'une multitude de facteurs : productivité des développeurs, richesse de l'écosystème, facilité de maintenance, exigences de sécurité, compatibilité avec les bibliothèques ou services tiers, ainsi que le niveau d'expertise disponible dans l'équipe. Cependant, à l'heure où les architectures distribuées s'imposent, où les micro-services sont invoqués des millions de fois quotidiennement, et où la facturation à la milliseconde est devenue la norme dans le cloud computing, l'impact énergétique devient un critère stratégique et rationnel, non plus seulement marginal ou théorique.

Un compromis pertinent consiste à adopter une architecture hybride. Les modules les plus critiques, en fréquence d'appel ou en consommation cumulée, peuvent être développés en langages sobres comme C, Rust ou Go, optimisant ainsi leur empreinte énergétique. Les parties de l'application moins sollicitées, ou qui exigent plus de flexibilité, peuvent quant à elles rester dans des langages de haut niveau comme Python ou Ruby, qui favorisent la rapidité de développement et l'agilité des itérations.

La différence fondamentale entre langages compilés et langages interprétés ou scriptés joue ici un rôle central. Les langages compilés comme C, Rust ou Go traduisent directement le code source en instructions machines optimisées, ce qui permet un contrôle fin sur l'allocation mémoire, l'optimisation des structures de données, et l'absence de surcouches coûteuses. À l'inverse, les langages de script, interprétés à l'exécution (Python, Ruby, PHP…), introduisent une latence structurelle, consomment davantage de ressources système et impliquent souvent des abstractions gourmandes en mémoire. Cela se traduit par une consommation énergétique nettement plus élevée, comme l'illustrent les écarts de performance relevés dans l'étude de 2017.

Un autre facteur déterminant est la présence ou non d'un garbage collector (ramasse-miettes), responsable de la gestion automatique de la mémoire. Si ce mécanisme améliore la sécurité mémoire et libère le développeur de certaines tâches complexes, il a un coût énergétique. Java, par exemple, dispose d'un garbage collector performant et optimisé depuis plusieurs années, ce qui explique sa relative bonne position dans le classement (1.98 fois la consommation du C, bien en deçà de Python ou Perl). Néanmoins, même les meilleurs *garbage collectors* induisent des surcoûts cycliques en traitement, liés aux phases de collecte, à la fragmentation mémoire ou à la gestion des objets éphémères. En revanche, des langages comme Rust permettent de gérer explicitement la mémoire sans garbage collector, grâce à un système de possession (*ownership*) et de vérification statique à la compilation, garantissant à la fois sécurité et efficacité énergétique.

Au total, le choix du langage ne peut plus s'abstraire de sa matérialité : mémoire allouée, temps machine utilisé, énergie consommée. Il s'agit moins de tout optimiser à outrance que de concevoir des systèmes conscients de leurs points névralgiques et capables de tirer parti des qualités propres à chaque langage selon leur fonction dans l'architecture. L'écoconception logicielle commence ainsi par un raisonnement stratégique sur les arbitrages techniques, et le langage de programmation en fait désormais pleinement partie.

Conclusion

Le grand mérite de cette étude est de démontrer, mesures à l'appui, que la consommation énergétique des logiciels n'est pas une abstraction théorique, mais une réalité quantifiable, et surtout, différenciée selon les choix techniques. Il devient donc nécessaire d'intégrer la dimension énergétique dès la phase de conception logicielle, non comme une contrainte supplémentaire, mais comme un levier d'optimisation globale.

Cela ne signifie pas qu'il faille tout réécrire en C ou en Rust, ni sacrifier la productivité au nom d'un idéal de performance brute. Il s'agit plutôt de repenser la structure des applications, d'identifier les points chauds en termes d'usage, et de réserver les langages les plus sobres aux modules les plus sollicités. Dans cette perspective, la sobriété énergétique rejoint des préoccupations déjà bien intégrées dans l'ingénierie logicielle : la sécurité, la maintenabilité, la résilience, ou encore l'accessibilité.

À mesure que le numérique devient un facteur majeur d'empreinte environnementale, en particulier dans les architectures cloud et distribuées, l'efficience énergétique des logiciels n'est plus une option, mais une responsabilité. Les développeurs, les architectes, les équipes DevOps ont désormais les outils pour raisonner en termes d'impact et arbitrer de manière éclairée. Choisir un langage, une architecture, une méthode de gestion mémoire, ce n'est plus seulement une affaire de performance ou de style : c'est aussi, déjà, un acte écologique.

Sources

L'article (doi) : https://doi.org/10.1145/3136014.3136031

L'article (PDF) : https://greenlab.di.uminho.pt/wp-content/uploads/2017/09/paperSLE.pdf

Données et sources : https://github.com/greensoftwarelab/Energy-Languages

Évaluation de la consommation énergétique des langages informatiques, Benoît Prieur, Devcon #19 - 100 % Green IT, Programmez!, en avril 2023, Lyon.

Réinventer la gestion des processus durables avec Appian et Xebia

Lamine Bensaid

Senior Solutions Architect, Appian
Expert en automatisation des processus métiers et transformation digitale, Lamine accompagne les grandes entreprises dans la conception et l'implémentation de solutions low-code évolutives sur Appian.

Jaideep Varier

VP Sales and Accounts - Intelligent Automation, Xebia, Spécialiste en intégration de systèmes et développement d'applications d'entreprise, Jaideep pilote des projets innovants à l'interface entre IT et enjeux métiers, notamment sur les sujets ESG et compliance.

L'intégration de l'intelligence artificielle dans la gestion des processus métiers (BPM) bouleverse la manière dont les entreprises abordent les enjeux ESG (Environnement, Social, Gouvernance). Face à la complexité croissante des exigences réglementaires, à la pression des parties prenantes et des défis techniques (collecte de données hétérogènes, complexité des standards, silos organisationnels), la plateforme ESG co-développée par Xebia et Appian propose une approche innovante. Cette plateforme modulaire, automatisée et extensible, pensée pour les développeurs et architectes SI permet d'automatiser, de centraliser et de fiabiliser la collecte, l'analyse et le reporting des données ESG, tout en restant extensible et personnalisable pour les développeurs.

Une architecture modulaire au service de la durabilité

La solution ESG conçue par Xebia repose sur une architecture modulaire et évolutive intégrée à Appian. Elle permet une centralisation des données, une automatisation des workflows, et une transparence accrue sur l'ensemble de la chaîne de valeur ESG. Cette plateforme se distingue par sa capacité à :

- S'adapter à différents cadres réglementaires (GRI, TCFD, etc.)
- Structurer les indicateurs de performance
- Intégrer des objectifs de développement durable de l'ONU
- Orchestrer les processus d'évaluation de matérialité
- Associer des données internes et tierces en temps réel

Les modules incluent notamment : la gestion des matérialisations, la collecte de données, les objectifs et KPI, la publication de rapports, et le benchmarking concurrentiel.

Fonctionnalités clés : de la collecte à la décision

1 Collecte de données multisources

La solution permet une collecte de données ESG provenant de sources multiples (internes, fournisseurs, partenaires). Des affectations automatiques aux départements concernés et des validations en chaîne garantissent la qualité et la cohérence des données.

2 Matérialité et structuration ESG

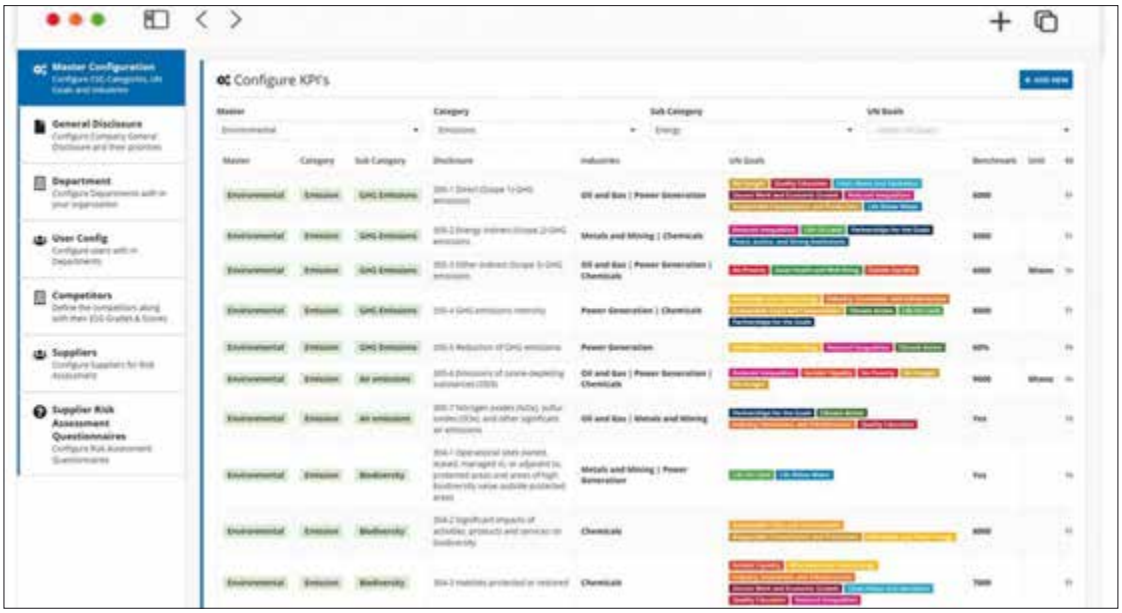
Les entreprises peuvent structurer leurs priorités via des analyses de matérialité dynamiques. Les parties prenantes sont impliquées tout au long du processus, ce qui facilite l'identification des facteurs clés et la définition des axes d'amélioration.

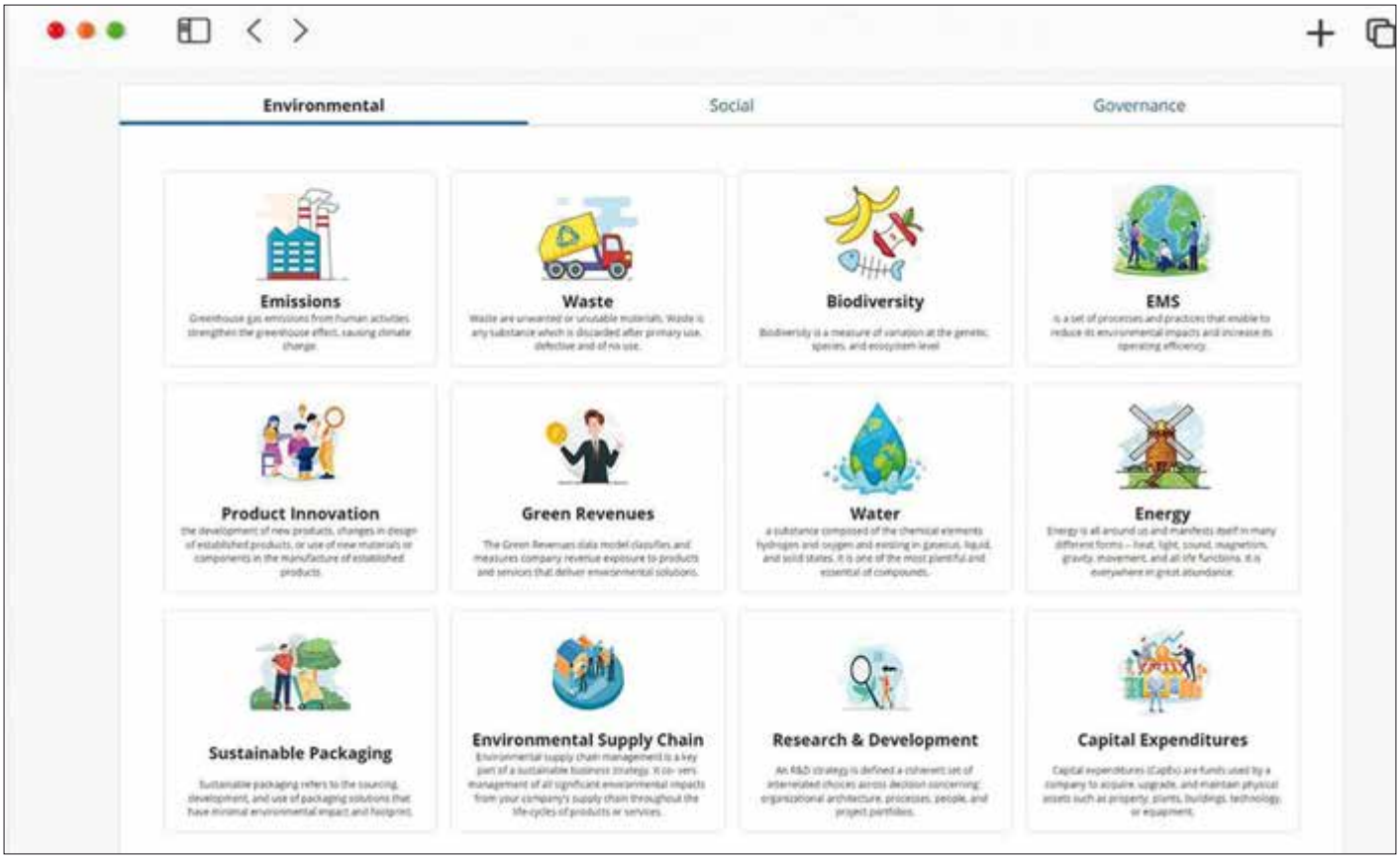
3 Objectifs et performances ESG

La plateforme permet de définir des objectifs mesurables alignés sur les standards internationaux. Un tableau de bord en temps réel offre une vision consolidée des progrès réalisés et alerte sur les retards.

4 Publication et conformité

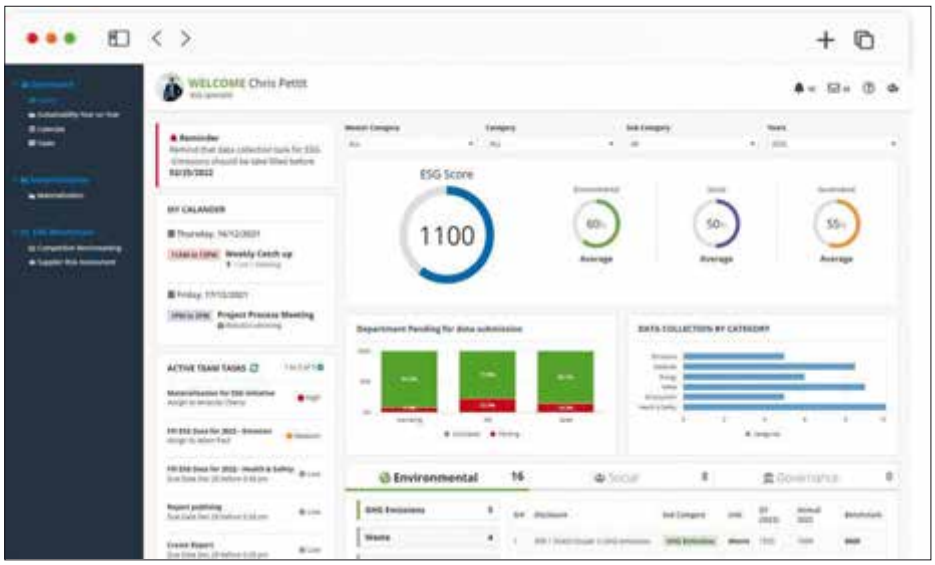
Grâce à la génération automatisée de rapports (intégrant les exigences réglementaires locales et internationales), les entreprises peuvent communiquer efficacement avec les régulateurs, les investisseurs et le grand public.





5 Évaluation du risque fournisseur et analyse concurrentielle

Un module permet de mesurer les performances ESG des fournisseurs, de piloter les plans de remédiation, et de se positionner par rapport à la concurrence sur des indicateurs clés.



Exemples de composants et scripts réutilisables

La plateforme inclut une série de composants personnalisables :

- **Formulaires dynamiques de saisie de KPI**
- **Générateur de rapport ESG multi-format**
- **Scripts d'intégration avec ERP / CRM (SAP, Salesforce)**
- **Dashboards interactifs 360°**
- **Configurations de disclosure selon GRI/ TCFD/CSRD**

Ces éléments peuvent être facilement réutilisés ou étendus dans d'autres projets Appian, ce qui accélère la mise en œuvre et réduit les coûts.

Conseils pour les développeurs et intégrateurs

Pour les développeurs souhaitant intégrer ou adapter ce type de solution dans un système d'information existant, voici quelques bonnes pratiques issues de l'expérience Xebia :

- **Anticiper l'orchestration des flux de données :** intégrer en amont les API nécessaires et les structures de données normalisées.
- **Travailler avec un framework ESG reconnu** dès le début du projet pour éviter des refontes coûteuses.
- **Modulariser le code et les interfaces** afin de faciliter la maintenance et l'évolution des cas d'usage.
- **Implémenter des logiques d'audit et de traçabilité dès le départ,** pour garantir la conformité réglementaire.
- **Collaborer avec les équipes métiers dès la phase de conception,** afin de refléter au mieux les processus réels et les contraintes réglementaires.

Conclusion

La solution ESG développée par Xebia sur la plateforme Appian est bien plus qu'un simple outil de reporting. Elle constitue une véritable boussole stratégique pour les entreprises souhaitant concilier performance et responsabilité. Intuitive, intelligente et rapide à déployer, elle transforme un sujet complexe et morcelé en un levier de compétitivité durable.

Jean-Pierre Roullin

Directeur des Systèmes d'information, découvrez pourquoi et comment les développeurs peuvent aussi agir. Alcatel-Lucent Enterprise

Green IT : réduire l'impact environnemental à la source avec un code responsable

Le Green IT fait souvent référence à la consommation des datacenters ou à la durabilité des terminaux tout au long de leur cycle de vie. Mais le rôle du code, des API et de l'architecture logicielle dans cette équation reste largement sous-estimé. Or, l'impact environnemental du numérique commence bien plus tôt qu'on ne le croit.

L'organisation The Shift Project estime qu'aujourd'hui, le numérique représente entre 3,5 et 4 % des émissions mondiales de gaz à effet de serre soit l'équivalent d'un pays de taille moyenne - en comparaison, celles de la France sont estimées à 4%. En cause : les serveurs, les terminaux et les réseaux. Mais les logiciels, les architectures et le code jouent un rôle tout aussi décisif dans l'empreinte carbone du digital.

Sur un site web, chaque clic génère de la pollution numérique. D'après l'outil en ligne Carbon Calculator une page web moyenne génère environ 1,76 gramme de CO_2 par chargement. Un site avec une audience mensuelle de 100 000 visites génère plus de 2 tonnes de CO_2 par an soit l'équivalent de 10 000 kilomètres en voiture thermique, si l'on se réfère aux calculs d'équivalence de l'ADEME. Bien que virtuelles, chaque ligne de code, chaque requête envoyée à un serveur ont donc un impact réel sur notre planète et peuvent conditionner, amplifier ou réduire l'impact écologique d'un service digital.

Différentes autres études réalisées par GreenIT.fr, Boavizta ou encore l'ADEME tendent, par ailleurs, à montrer que la consommation énergétique liée à la conception logicielle peut varier de 1 à 8 pour un même service, selon l'optimisation du front, du back et de l'infrastructure réseau. En d'autres termes, des choix purement logiciels peuvent multiplier l'impact environnemental d'un service sans pour autant changer sa valeur perçue par l'utilisateur final.

De l'importance de penser "réseau" dès le processus de développement

D'après les données de Boavizta, une application mobile effectuant un rafraîchissement automatique toutes les 30 secondes peut générer jusqu'à 3 Go de trafic mensuel par utilisateur — un volume qui pourrait être divisé par 10 avec un simple système de mise en cache intelligent ou une stratégie de synchronisation événementielle.

Au cours du processus de développement, l'infrastructure réseau est, en effet, souvent négligée : appels API trop fréquents, requêtes inutiles, absence de mise en cache, contenus non compressés, etc. Tous ces choix ont un coût énergétique bien réel, même s'il reste invisible dans une console. Une requête API vers un service cloud peut consommer de 0,02 à 0,1 gCO_2e *(unité mesurant les grammes d'équivalent dioxyde de carbone)* selon sa fréquence, son poids, le type de réseau utilisé (4G, Wi-Fi, fibre), comme le montre certaines modélisations. Cela paraît minime mais multiplié par des milliers d'utilisateurs, plusieurs fois par jour, cette consommation grimpe à vitesse exponentielle.

Mais bonne nouvelle, des leviers concrets existent. Réduire la taille des payloads dans les réponses API (afin d'éviter le sur-retour inutile), mettre en cache côté client quand cela pertinent, utiliser la compression HTTP systématiquement, optimiser la fréquence des appels (éviter les "pings" constants pour synchroniser des données rarement modifiées) ou encore adopter une logique "offline first" ou edge computing pour limiter le nombre de requêtes cloud sont tout autant de gestes simples, souvent déjà connus et que les développeurs ont la possibilité de systématiser.

Outils de collaboration : le poids caché de la "productivité numérique"

Autre zone grise : les outils de communication et de collaboration, souvent considérés comme "virtuels" mais à l'impact très concret. En effet, selon l'Agence Internationale de l'Énergie, une heure de visioconférence HD consomme jusqu'à 1 000 Wh en fonction de la résolution et la qualité de la connexion soit l'équivalent d'un réfrigérateur de classe A++ en 24h. Et si l'on désactive la vidéo pour passer en mode audio-only, cette consommation peut être divisée par 6 à 10, selon une étude de Purdue University (2021). Les services de messagerie, de visio ou de téléphonie IP, permettent peut-être de limiter les émissions liées aux déplacements physiques mais ne sont finalement pas pour autant neutres.

En tant que concepteurs de ces outils, les développeurs ont la responsabilité d'y intégrer des modes "sobres par défaut" : activation automatique de la caméra désactivée si inutilisée, résolution adaptative intelligente selon les conditions réseau, compression audio/vidéo optimisée, paramètres utilisateurs orientés sobriété (modes audio-only, mise en veille automatique, etc.). Au-delà des aspects "techniques", la manière dont ces outils sont utilisés (réunions inutiles, messages redondants, flux constants) appelle aussi à une hygiène numérique collective : concevoir des outils moins gourmands, c'est aussi encourager des usages plus responsables.

Vers un Green DevOps ?

Si le Green IT est souvent l'affaire de la DSI ou des équipes infra, les équipes de développement peuvent elles aussi s'approprier pleinement cet enjeu. Des outils commencent d'ailleurs à apparaître dans les chaînes CI/CD, c'est à dire dans l'ensemble des étapes automatisées permettant de construire, tester, et déployer du code logiciel de manière continue et rapide.

Citons, entre autres, les outils, Open Source ou non, suivants : EcoCode (extension pour SonarQube pour identifier les « green code smells » — des structures de code susceptibles d'augmenter la consommation énergétique sans valeur ajoutée fonctionnelle), GreenFrame (évaluation de l'empreinte d'un parcours utilisateur), Scaphandre (mesure de la consommation énergétique des processus), Cloud Carbon Footprint (mesure de l'empreinte carbone liée à la consommation des Cloud AWS, Azure, GCP), Green Metrics Tool (évaluation de l'impact environnemental d'un site web ou d'une application en termes de consommation de GES, eau et autres ressources). Les développeurs disposent aussi d'outils tels que Lighthouse + EcoIndex, le pendant "vert" de Lighthouse de Google, la Base Empreinte® de l'ADEME ou encore Boavizta.

Autre initiative prometteuse : le Référentiel Général d'Écoconception de Services Numériques de la Direction Interministérielle du Numérique. S'appuyant sur des mesures issues du Guide de Référence de Conception Responsable de Services Numériques, de l'Institut du Numérique Responsable, il propose 79 bonnes pratiques adaptées aux projets numériques.

En parallèle, le législateur se saisit aussi de ce sujet. La loi de Réduction de l'Empreinte Environnementale du Numérique impose désormais plusieurs standards en matière de durabilité technologique, dont la définition d'une stratégie d'éco-conception digitale pour les collectivités locales françaises de plus de 50 000 habitants. Le Règlement européen sur l'écoconception des produits durables, établit, quant à lui, un cadre pour l'écoconception des produits. La performance énergétique des logiciels n'est pas un critère mentionné spécifiquement mais y contribue largement.

Coder de manière responsable : des gestes simples et à impact

Coder de manière responsable revient mettre en place de bonnes pratiques, connues mais sous-exploitées, dans ses choix de développement pour éviter des traitements inutiles, favoriser l'inclusivité et optimiser les ressources dès la première ligne et cela sans nuire à l'expérience utilisateur.

Concrètement, cela signifie :

- Charger les images uniquement lorsqu'elles deviennent visibles (<img loading="lazy">),
- Utiliser des polices système pour éliminer les requêtes tierces superflues,
- Minifier CSS et JavaScript afin de réduire les octets transférés à chaque requête,
- Éviter les traitements redondants, par exemple en stockant array.length dans une variable lors des boucles,
- Ajouter des attributs aria-* pour améliorer la navigation pour des millions d'utilisateurs sans coût technique supplémentaire,
- Ou encore remplacer des bibliothèques lourdes comme Moment.js par les APIs natives de Date.

Chaque décision compte : 100 Ko en moins, ce sont des milliers de Mo économisés à l'échelle annuelle pour un site fréquenté. Ce sont ces gestes, souvent invisibles à l'œil nu, qui jettent les jalons d'un numérique plus sobre, plus inclusif et durable.

Vers un futur écologique du numérique avec un code éco-responsable

Les objectifs de réduction d'empreinte carbone ne pourront être atteints sans intégrer pleinement les enjeux environnementaux du cycle de vie logiciel. Tout comme la sécurité, la performance ou l'accessibilité, la sobriété numérique est un critère indiscutable.

Le code a un poids. Et nous devons nous donner les moyens de le rendre plus léger.

EXEMPLES DE BONNES PRATIQUES DE CODAGE RESPONSABLE

Éviter les calculs redondants

Avant :

```
// Inefficace : recalcul de la longueur à chaque itération
for (let i = 0; i < items.length; i++) {
 process(items[i]);
}
```

Après :

```
// Efficace : stockage préalable
const len = items.length;
for (let i = 0; i < len; i++) {
 process(items[i]);
}
```

Lazy loading des images

Avant :

```
<!— Chargement immédiat de toutes les images —>
<img src="photo.jpg" alt="Description utile" />
```

Après :

```
<!— Chargement différé uniquement quand visible —>
<img src="photo.jpg" loading="lazy" alt="Description utile" />
```

Réduction des dépendances lourdes

Avant :

```
// Utilisation d'une bibliothèque lourde pour manipuler une date
import moment from 'moment';
const now = moment().format('YYYY-MM-DD');
```

Après :

```
// Utilisation des API natives
const now = new Date().toISOString().split('T')[0];
```

Fonts système au lieu de polices externes

Avant :

```
/* Téléchargement d'une police personnalisée */
@import url('https://fonts.googleapis.com/css?family=Roboto');
body { font-family: 'Roboto', sans-serif; }
```

Après :

```
/* Police système déjà disponible sur l'appareil */
body {
 font-family: -apple-system, BlinkMacSystemFont, "Segoe UI", Roboto,
 sans-serif;
}
```

Mise en cache d'une réponse API

Avant :

```
// Appel API systématique à chaque chargement
fetch('/api/data').then(handleData);
```

Après :

```
// Cache des données localement quand possible
const cached = localStorage.getItem('data');
if (cached) {
 handleData(JSON.parse(cached));
} else {
 fetch('/api/data')
  .then(res => res.json())
  .then(data => {
   localStorage.setItem('data', JSON.stringify(data));
   handleData(data);
  });
}
```

Accessibilité sans coût supplémentaire

Après :

```
<!— Amélioration immédiate de l'accessibilité —>
<button aria-label="Fermer le menu">
 <svg>...</svg>
</button>
```

Compression automatique via HTTP

Activez systématiquement la compression *GZIP* ou *Brotli* sur le serveur (Nginx, Apache, CDN) pour réduire le volume de données transférées.

L'application mobile Espace Client Bouygues Telecom récompensée pour sa démarche de sobriété numérique par Greenspector

François Lankar

Responsable maîtrise d'œuvre pour l'espace client mobile et l'assistance web chez Bouygues Telecom

François, passionné par le digital et les projets qui font bouger les choses.
Depuis 6 ans, je travaille chez Bouygues Telecom en tant que Responsable MOE pour l'espace client mobile et l'assistance web. Mon rôle ? Coordonner les équipes, piloter les évolutions techniques et améliorer l'expérience utilisateur au quotidien. J'aime quand la tech est au service du concret, et surtout quand elle simplifie la vie des clients.

L'application mobile Espace Client Bouygues Telecom a obtenu cette année le Certificat de Sobriété Numérique niveau argent, avec une note de 73/100, décerné par Greenspector, entreprise à mission éditrice de la suite Greenspector Studio. Cette reconnaissance valorise les efforts de la DSI pour réduire l'impact environnemental de nos services numériques tout en offrant le meilleur service à nos clients. Une empreinte carbone difficile à mesurer, car portée par les usages des clients sur leur propre matériel, et pourtant non-négligeable. Grâce aux optimisations mises en œuvre, les émissions sont passées de 14,2 gCO2eq à 9 gCO2eq sur les parcours clients évalués sous Android, soit de 1576 tonnes CO_2eq par an au total à 883 tonnes CO_2eq par an. Une économie annuelle estimée à 693 tonnes de CO_2eq, soit l'équivalent de 392 allers-retours Paris-New York en avion*.

Du bronze à l'argent : un processus d'amélioration continue depuis 2022

Déjà certifiée au niveau bronze en 2022, l'application a franchi un nouveau cap pour obtenir la certification argent. Grâce aux données fournies par Greenspector Studio sur plusieurs cycles d'analyse et de feedback, nous avons pu identifier les parcours les plus consommateurs en énergie et ressources et cibler les optimisations nécessaires. L'objectif : améliorer la performance, réduire la consommation de données et renforcer l'efficacité énergétique de l'application Bouygues Telecom. Un projet qui a nécessité près d'un an de travail, dans une démarche dynamique suivant les principes de l'écoconception.

Une analyse approfondie à partir des rapports de Greenspector

Les données mesurées par Greenspector Studio donnent des indications sur la consommation d'un parcours utilisateurs sur un terminal réel, avec des mesures au niveau de chaque étape du parcours (action de l'utilisateur, chaque changement de vue, pause…). Pour les étapes les plus consommatrices, nous avons ensuite utilisé deux outils de profilage pour identifier les raisons des surconsommations et des pistes d'amélioration.

Par exemple, pour les données mobiles, l'analyse nous donnait initialement un score de 44.

Suite aux optimisations apportées, l'analyse donne un score de 75.

Tout d'abord l'outil de développement Xcode nous a permis d'analyser le trafic HTTP sur les pages identifiées. Nous avons ainsi pu observer le comportement des requêtes GraphQL et des appels API, ce qui nous a aidé à repérer les inefficacités dans la gestion des échanges de données. Au lieu d'utiliser plusieurs appels GraphQL pour chaque page, nous avons également regroupé certaines requêtes dans notre requête globale, qui est chargée une seule fois et conservée en cache, réduisant ainsi la latence et l'utilisation des ressources réseau.

Enfin, nous avons utilisé des outils de développement spécifiques pour inspecter la performance de nos composants React. Ils nous ont permis de modifier les accessoires et l'état des composants directement, facilitant ainsi l'identification des problèmes de performance. En inspectant le renderer, nous avons pu déterminer quels composants impactaient le plus les performances et adapter nos optimisations en conséquence. Nous avons par exemple simplifié et amélioré le rendu des pages, réduisant la consommation de mémoire et augmentant la vitesse de traitement.

Des optimisations techniques, pour améliorer l'efficacité énergétique de l'application et l'expérience utilisateur

Une fois l'analyse faite, nous avons ensuite procédé à différentes optimisations techniques. Pour améliorer la rapidité et la consommation de données de l'application, nous avons optimisé la gestion du cache grâce à l'option `unmountOnBlur: false` dans la configuration de navigation de React Navigation.

```
  <AuthenticatedBottomTab.Navigator screenOptions={bottomTapOptions}
tabBar={(props) => <TabBar {...props} />}>
    <AuthenticatedBottomTab.Screen
     component={isAllContractsCancelled ? CancelledCustomerHomeScreen :
HomeScreen}
     initialParams={{ name: BottomTapScreens.HOME_SCREEN }}
     key={BottomTapScreens.HOME_SCREEN}
     name={BottomTapScreens.HOME_SCREEN}
    options={{ tabBarLabel: t('common.home'), title: IconName.HOME,
unmountOnBlur: false }}
   />
```

Grâce à cette optimisation, les utilisateurs bénéficient d'une navigation plus fluide et rapide car les composants restent montés lors de la navigation entre les différentes sections de l'application. Cela réduit aussi la consommation de bande passante, un facteur important pour l'efficacité énergétique de l'application Bouygues Telecom.

Nous avons également travaillé à l'amélioration globale des performances de l'application. Pour cela, nous avons activé l'option EnableScreen proposée par React Native, qui optimise l'activation des écrans en fonction de leur visibilité par l'utilisateur. Ainsi, l'application gère les transitions entre les écrans de manière plus efficace, réduisant la charge de calcul que le processeur doit gérer à chaque changement de vue.

```
import { enableScreens } from 'react-native-screens'
import { is } from 'superstruct'
import App from './App'
import { handleNotif, handleNotifAction, sendDeeplinkFromNotif } from './libs'
import { airshipAlternativeResponseNotif, airshipDeeplink, airshipResponse
Notif } from './superstructSchema'
import type { PartialAppContext } from './types/Notification'
import '@bytel/invoices-core'
const appName = 'mcare'
if (Platform.OS === 'android') {
 enableScreens()
}
```

Nous avons par ailleurs utilisé les hooks `useCallback` et `useMemo`. En stabilisant les fonctions au sein des composants, `useCallback` réduit la charge de traitement, en évitant des ré-rendus inutiles qui peuvent alourdir l'application. En limitant la fréquence des recalculs, `useMemo` aide quant à lui à réduire la consommation énergétique. Ces deux hooks contribuent à une meilleure efficacité énergétique de l'application Bouygues Telecom.

```
const updateStatus = useCallback(() => {
  checkNotifications().then(async (result) => {
    const isGranted = result.status === STATUS.GRANTED
    setStatus(result.status)
    if (isGranted) {
      await notifSubscribe()
      Airship.push.setUserNotificationsEnabled(true)
      await AsyncStorage.setItem('permissionNotif', JSON.stringify(true))
      return
    }
    Airship.push.setUserNotificationsEnabled(false)
    return await AsyncStorage.setItem('permissionNotif', JSON.stringify
(false))
  })
 }, [])

 const dataKey = useMemo(() => {
  if (is4Gor5GKey(currentContract)) {
    return 'key45G'
  } else if (isMobile(currentContract)) {
    return 'mobile'
  } else if (isBBox(currentContract)) {
    return 'bbox'
  } else {
    return 'key55Gbox'
  }
 }, [currentContract])

 const dataCards = useMemo(() => NeedHelpData[dataKey], [dataKey])
 const helpCards = useMemo(() => dataCards.slice(0, -1), [dataCards])
 const lastHelpCard = useMemo(() => dataCards[dataCards.length - 1],
[dataCards])
```

Pour améliorer le temps d'affichage de certaines sections, nous avons enfin utilisé la technique « Renderer after Interaction » en intégrant la méthode `InteractionManager .runAfterInteractions` de React Native. Cette technique permet de différer le rendu de certains composants jusqu'à ce que toutes les interactions utilisateur soient traitées. En priorisant les interactions utilisateur avant le rendu des composants lourds, cette technique permet d'améliorer la réactivité globale de l'application. Les utilisateurs ressentent une fluidité accrue, surtout sur les appareils où les ressources sont limitées.

```
useEffect(() => {
  InteractionManager.runAfterInteractions(() => {
    !isRendering && setIsRendering(true)
  })
 }, [])

 return (
  <GreyBackground>
    <MainHeaderTitle title={t('modules.myAccount.title')} />
    <ScrollView ref={scrollViewRef} showsVerticalScrollIndicator={false}
testID='MyAccountScreenScrollView'>
      <CustomSection backgroundColor={TrilogyColor.NEUTRAL_FADE}
paddingless={['TOP']}>
        <FullNameAndTagHolder />
        <FirstSection sendTagClic={sendTagClic} />
        <SecondSection sendTagClic={sendTagClic} />
        <ContactUs />
        <SkeletonScreen isRendering={isRendering} />
        <FavoriteShopBox isRendering={isRendering} />
        <SettingsAndAboutUs isRendering={isRendering} />
        <DisconnectAndDeveloper isRendering={isRendering} />
        <Spacer size={SpacerSize.TWO} />
        <AppVersion isRendering={isRendering} />
      </CustomSection>
    </ScrollView>
  </GreyBackground>
 )
```

Une démarche plus responsable et intégrée

Greenspector Studio est désormais intégré à notre chaîne d'intégration continue. Chaque nouvelle version de l'application est analysée pour éviter toute régression. Les étapes les moins performantes sont identifiées et des tickets d'amélioration sont créés dans une logique d'écoconception continue du service numérique.

Cette démarche est une illustration concrète de notre ambition d'accélérer la décarbonation de des activités de Bouygues Telecom, figurant dans l'axe environnemental de sa stratégie RSE « Agir ensemble pour un numérique plus responsable, positif et inspirant ».

**Source : impactco2.fr*

Les 10 bonnes pratiques d'écoconception web pour débuter

Florilège des bonnes pratiques d'écoconception à mettre en œuvre en priorité lorsqu'on commence à s'intéresser à ce sujet et qu'on a un profil technique.

Frédéric Bordage
fondateur du collectif Green IT et initiateur du Référentiel d'écoconception web en 2011.

Entre 1995 et 2025, le poids d'une page web a été multiplié par 190 ! Cette inflation se traduit par une durée de vie toujours plus courte de nos terminaux. Celle d'un ordinateur a ainsi été divisée par 3 entre 1995 et 2025. Or, comme c'est leur fabrication qui concentre les impacts environnementaux, le numérique représente aujourd'hui 40 % du budget annuel soutenable d'un européen. C'est 10 fois trop par rapport aux objectifs de développement durable que l'humanité s'est fixés. Heureusement, grâce à l'écoconception, nous avons un formidable effet de levier à notre disposition. En appliquant les 10 bonnes pratiques clés que nous détaillons dans cet article, vous pourrez contribuer à diviser par 2, 4 et même jusqu'à 8 les impacts environnementaux nécessaires pour délivrer exactement le même service en ligne : trouver l'horaire d'un train, consulter le solde de son compte en banque, prendre rendez-vous chez le médecin, etc.

Cette sélection de bonnes pratiques est dédiée au profil spécifique (technique) des lecteurs et lectrices de Programmez!. Elles sont issues de la 5ème édition du Référentiel d'écoconception web (RWEB) publié depuis 13 ans par le collectif Green IT. Gratuites et sous licence Creative Commons, ces bonnes pratiques vous aident à réduire les impacts environnementaux de vos services en ligne tout en améliorant leur performance et l'expérience utilisateur. Alors pourquoi s'en priver !

Prêt.e à agir concrètement ? C'est parti !

1 Choisir les technologies les plus adaptées (#67)

Le choix des technologies est primordial pour réduire la consommation de ressources (RAM, CPU, etc.). Pour un site dont le contenu change peu, on privilégiera des pages HTML statiques tandis que pour une fonction métier transactionnelle (prise de rendez-vous, horaire d'un train, demande de devis, etc.) on préférera une SPA (Single Page Application) pour limiter les allers-retours avec le serveur et rendre l'expérience utilisateur plus fluide. Dans la même logique, le choix d'un framework JS léger est souvent préférable.

Figure A

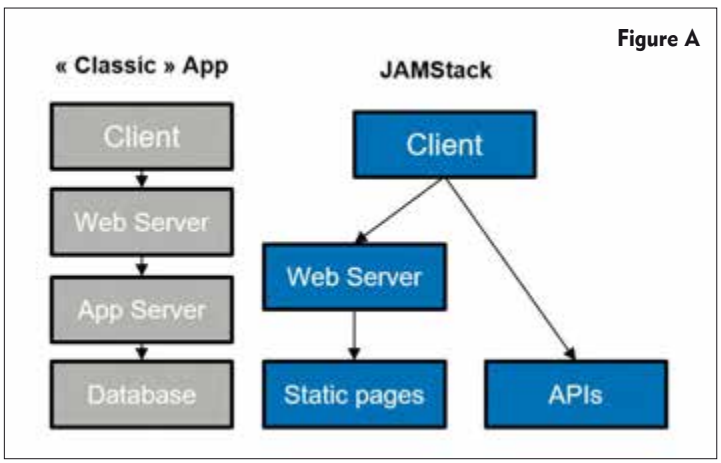


Illustration : https://drive.google.com/file/d/10qdwiZgDvdbmeIGI9DyCfmI2q6PyBYPz/view?usp=sharing
source : Zenika avec Jamstack.org

2 Favoriser les pages statiques (#18)

Dès que c'est possible, il faut privilégier du contenu statique. Pour une landing page ou un simple site vitrine, on s'appuiera sur HTML, CSS et JS. On générera un blog avec peu d'activité avec un jamstack (Jekyll, Hugo, Gatsby, Eleventy, etc.) qu'on pourra administrer, si nécessaire, avec via un CMS headless ou un CMS headless flat-file (Strapi, Contentful, Flextype, Cecil.app, etc.). Pour les rares portions dynamiques, on peut compléter avec un framework adapté type Next.js,Nuxt.js, Svelte, etc. **Figure A**

3 Ne charger que le code / données indispensable (#45)

Pour des raisons de performance perçue, il peut être tentant de précharger des ressources qui pourraient être utilisées si une action spécifique était effectuée par l'utilisateur (y compris une navigation vers une autre page). Mais si cette action n'est pas entreprise, ces pré- chargements n'auront servi qu'à gaspiller des ressources. Il faut donc éviter des instructions telles que <link rel="prerender">, ainsi que tous les rel="preload", rel="prefetch", rel="preconnect", rel="modulepreload" et rel="dns-prefetch" dont la ressource ne sera pas utilisée sur la page courante, ou utilisée uniquement sous certaines conditions.

4 Éviter la lecture et le chargement automatique des vidéos et des sons (#106)

Le préchargement (*preload*) et l'activation automatique des vidéos et des sons (*autoplay*) engendre une sur-utilisation inutiles des ressources (RAM, CPU, bande passante) tant sur le terminal que côtés serveurs et réseaux. Il ne faut donc pas ajouter l'attribut autoplay dans les balises <video> ou <audio> et définir la valeur de l'attribut *preload* des balises <video> ou <audio> à none comme dans cet exemple <video src="fichiervideo.webm" preload="none"></video>

5 Limiter le nombre de requêtes HTTP (#46)

Avec certains serveurs web, plus le nombre de requêtes par page est important et moins vous pourrez servir de pages par serveur, ce qui augmente mécaniquement le nombre de serveurs nécessaires. Plusieurs techniques peuvent être mises

en place pour réduire le nombre de requêtes par page : concaténer les fichiers JS et CSS, utiliser le *lazyload* pour les images, ou bien encore créer des sprites ou des fonts d'icônes pour les icônes.

Par exemple, les coins arrondis des cases peut être gérés en CSS (1 requête) plutôt qu'avec des images (autant de requêtes que d'images) :

```
#cadre {
  border-radius: 10px;
}
<div id="cadre">
  <p>
   Lorem ipsum dolor sit amet
  </p>
</div>
```

6 Eviter les blocages dus aux traitements javascript trop longs (#52)

Pour construire la structure (*DOM*) d'une page et l'afficher une page, le navigateur analyse le code HTML et exécute le code CSS et JavaScript. Plus les traitements javascript sont gourmands et longs et plus l'internaute aura une impression de faible performance ce qui le poussera à changer d'ordinateur ou de smartphone. Pour éviter cela, découpez vos codes JavaScript en petites tâches exécutées au moment opportun et non pas avant le chargement complet de la page.

7 Mettre en cache les données calculées souvent utilisées (#16)

Lorsque des calculs de valeurs ou de données sont coûteux en ressources, il faut les mettre en cache afin de ne pas réitérer ces opérations. Les systèmes de cache de type key-value store sont prévus pour stocker ces données. Généralement montés entièrement en mémoire vive (RAM), ils génèrent d'importantes économies de cycles CPU si les données calculées sont très souvent sollicitées. Par exemple, les jetons d'accès en OAuth2 sont associés à une date d'expiration. Mettre en cache le jeton et son délai d'expiration évite des appels inutiles au serveur d'autorisation et de devoir revalider le jeton.

8 Choisir un format de données adapté (#63)

Le type utilisé pour manipuler et stocker une donnée a un impact significatif sur la consommation mémoire et les cycles processeurs à tous les niveaux : base de données, serveur d'applications et même dans le navigateur (manipulation via JavaScript). Il faut donc les choisir avec sagesse. Par exemple, pour stocker des nombres compris entre 1 et 127, un TINYINT nécessite 8 fois moins d'octets (1) qu'un BIGINT (8 octets). Multiplié par des millions de requêtes, cela fait une sacrée différence !
Dans l'exemple ci-dessous, le nombre caractères et le type d'entier est défini en fonction de données statistiques :

```
CREATE TABLE utilisateur
(
  id INT PRIMARY KEY NOT NULL,
  nom VARCHAR(30),
  prenom VARCHAR(30),
  email VARCHAR(50),
  date_naissance DATE,
  pays VARCHAR(50),
)
```

ou

Par exemple, préférez un SMALLINT à un VARCHAR() pour stocker un code postal :

```
CREATE TABLE utilisateur
(
  id INT PRIMARY KEY NOT NULL,
  nom VARCHAR(30),
  prenom VARCHAR(30),
  ~~code_postal VARCHAR(5),~~
  code_postal SMALLINT,
)
```

9 Compresser les fichiers texte : CSS, JS, HTML et SVG (#76)

Pour limiter l'utilisation de la bande passante et améliorer les temps de chargement, compressez les fichiers texte : CSS, JS, HTML et SVG. Par exemple, pour utiliser BROTLI dans NGinx, il suffit d'ajouter la configuration suivante :

```
# for compressing responses on-the-fly
load_module modules/ngx_http_brotli_filter_module.so;
# for serving pre-compressed files
load_module modules/ngx_http_brotli_static_module.so;
http {
   server {
   brotli on;
   #...
   }
}
```

10 Économiser la bande passante grâce aux Service Workers

La plupart des pages partagent une structure commune encadrant le « contenu utile ». On peut donc assez facilement les reconstituer par concaténation de trois ressources : un en-tête et un pied de page commun à toutes les pages, et le contenu propre à chacune. Cette

10 BONNES PRATIQUES D'ÉCOCONCEPTION POUR COMMENCER

1. Choisir les technologies les plus adaptées (#67)
2. Favoriser les pages statiques (#18)
3. Ne charger que le code / données indispensable (#45)
4. Éviter la lecture et le chargement automatique des vidéos et des sons (#106)
5. Limiter le nombre de requêtes HTTP (#46)
6. Eviter les blocages dus aux traitements javascript trop longs (#52)
7. Mettre en cache les données calculées souvent utilisées (#16)
8. Choisir un format de données adapté (#63)
9. Compresser les fichiers texte : CSS, JS, HTML et SVG (#76)
10. Économiser la bande passante grâce aux Service Workers (#60)

concaténation peut s'effectuer directement dans le navigateur via un Service Worker (un fichier JS) qui joue le rôle de proxy entre le navigateur et le serveur. Avec l'en-tête et le pied de page mis en cache HTTP, on ne télécharge plus que le « contenu utile ». On concède ici un peu de temps processeur au niveau du terminal, une requête initiale supplémentaire pour le Service Worker, puis deux requêtes supplémentaires pour l'en-tête et le pied de page, contre de grosses économies concernant les quantités de données échangées par la suite, ainsi qu'un gain significatif sur le temps de chargement des pages.

Exemple de mise en cache des ressources gérées par le service worker :

```
const addResourcesToCache = async (resources) => {
 const cache = await caches.open("v1");
 await cache.addAll(resources);
};

self.addEventListener("install", (event) => {
 event.waitUntil(
  addResourcesToCache([
   "/",
   "/index.html",
   "/style.css",
   "/app.js",
   "/image-list.js",
   "/star-wars-logo.jpg",
   "/gallery/bountyHunters.jpg",
   "/gallery/myLittleVader.jpg",
   "/gallery/snowTroopers.jpg",
  ]),
 );
});
```

Source : Mozilla Developper Network

Gagnez un exemplaire de la nouvelle édition du livre "écoconception web : les 115 bonnes pratiques"

Votre magazine préféré et le collectif Green IT se sont associés pour organiser un jeu-concours vous permettant de gagner un exemplaire papier de la nouvelle édition du livre *Ecoconception web : les 115 bonnes pratiques*. Pour jouer, c'est très simple : il vous suffit de vous inscrire et de répondre à une question dont la réponse est présente dans cet article.

Je m'inscris et je joue —>

Les partenaires 2025 de

programmez!

Le magazine des dév - CTO & Tech Lead

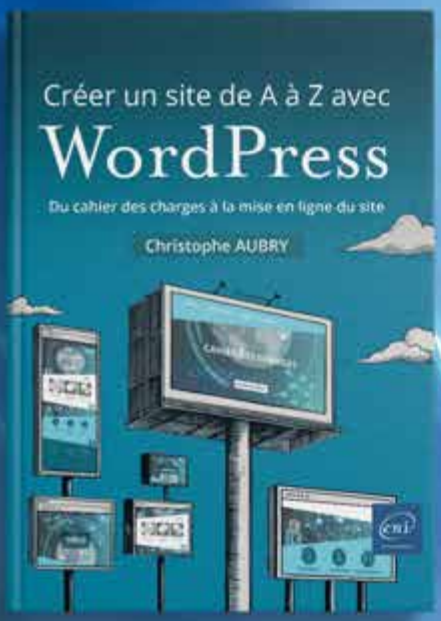
Vous voulez soutenir activement Programmez! ?
Devenir partenaires de nos dossiers en ligne et de nos événements ?

Contactez-nous dès maintenant :

ftonic@programmez.com

APPRENEZ
PROGRESSEZ
MAÎTRISEZ

avec nos nouveaux livres



ENI Editions

FORMATIONS À L'INFORMATIQUE EN LIGNE

www.editions-eni.fr