

L'Architecture Technique : les services de plate- forme

Architecture d'Entreprise

L'Architecture Technique : les services de plate-forme

Architecture d'Entreprise

III — LE CARACTÈRE

« The Technical Reference Model (TRM) is a reference model that provides a vocabulary and a structure for describing technical components. » — The Open Group, TOGAF Series Guide — TRM (G229)

« Une bonne architecture technique n'est pas celle qui choisit les bons outils. C'est celle qui peut en changer sans tout réécrire. »

Vidéo d'introduction

 Télécharger cette leçon en PDF

Le datacenter qu'on n'arrivait plus à éteindre

En 2020, Mapify — éditeur SaaS de cartographie pour les collectivités — fait face à une croissance subite. 200 % d'augmentation du trafic en 18 mois, liée à une série de contrats publics. La plate-forme, conçue en 2018 pour 10 000 utilisateurs simultanés, est dimensionnée pour 50 000 — mais les incidents se multiplient. Trois coupures majeures en six mois, dont une de 14 heures qui a fait la une de la presse spécialisée.

L'équipe technique, brillante mais submergée, a fait ce que beaucoup d'équipes brillantes font dans cette situation : **empiler les solutions ponctuelles**. Load balancer supplémentaire devant le load balancer existant, base de données répliquée à la hâte sur un autre datacenter, services de cache improvisés. Six mois plus tard, personne — y compris les architectes — n'a une vision claire de l'architecture technique en production. La documentation est obsolète. Les diagrammes sont des souvenirs.

C'est à ce moment qu'Antoine, l'architecte technique, est recruté. Son premier acte est contre-intuitif : **ne pas changer l'infrastructure existante**. Son deuxième acte l'est tout autant : **représenter l'architecture technique telle qu'elle est vraiment, pas telle qu'elle était censée être**.

Antoine passe un mois à interviewer les équipes, à examiner les configurations réelles (pas celles documentées), à reconstituer la **stack technique effective** — serveurs, réseaux, stockage, middleware, services d'observabilité. Le résultat est un **Technical Reference Model (TRM)** adapté à Mapify : un modèle de référence qui décrit ce qui existe, avec ses forces et ses fragilités.

Une fois le TRM posé, Antoine peut alors distinguer : - **Les services de plate-forme durables** — les composants qu'il faut investir, à stabiliser, à gouverner - **Les composants à risque** — ceux qui marchent par chance ou par héroïsme individuel, et qu'il faut reprendre - **Les candidats au décommissionnement** — ceux qui ne servent plus, mais continuent à coûter

Le TRM est la première étape d'une architecture technique qui **décrit l'existant** avant de prescrire la cible.

C'est ce que cette leçon cherche à vous transmettre : **comment le TRM rend visible l'architecture technique effective, et comment il sert de base à toute décision de plate-forme**.

Le TRM de Mapify illustre les sept catégories de services techniques que toute organisation doit cartographier — avec, au milieu, la plate-forme qui les relie :

Premier principe — Le TRM décrit ce qui est, pas ce qu'on voudrait

Le TRM (Technical Reference Model) est un modèle de référence qui décrit la stack technique effective d'une organisation — pas la stack idéale. Cette distinction est cruciale : sans description honnête de l'existant, toute décision de cible est construite sur du sable.

L'Open Group publie un TRM générique qui identifie les grandes catégories de services techniques :

- **Services applicatifs** — l'exécution des processus métier
- **Services de données** — la persistance, l'indexation, la réplication
- **Services d'infrastructure** — serveurs, stockage, réseau, virtualisation
- **Services de plate-forme** — middleware, orchestration, service mesh
- **Services de sécurité** — identité, authentification, chiffrement, audit
- **Services d'observabilité** — monitoring, logging, tracing, alerting
- **Services de déploiement** — CI/CD, infrastructure as code, automatisation

Le TRM générique est utile comme **grille de lecture** — il permet de cartographier l'existant dans chaque catégorie. Mais le TRM **spécifique** à l'organisation est ce qui compte : il décrit les produits concrets utilisés (AWS EC2, PostgreSQL 15, Istio 1.20, etc.), avec leur criticité, leur dette, leur risque de fin de support.

La contrefaçon classique consiste à dessiner un TRM idéal (Kubernetes partout, microservices, observabilité full-stack) sans décrire l'existant. Cette fiction rend les slides impressionnantes mais les décisions intenable.

***Et toi ?** As-tu un TRM documenté pour ton organisation ? Décrit-il l'existant ou la cible ? Quelle est la dernière date de mise à jour ?*

Deuxième principe — La plate-forme abstraite rend la migration possible

Une plate-forme abstraite n'est pas une plate-forme Kubernetes ou AWS — c'est une couche d'abstraction entre les applications et l'infrastructure sous-jacente. Cette abstraction permet de migrer sans réécrire.

L'Open Group recommande que les applications soient déployées sur des **plate-formes abstraites** — des couches logicielles qui masquent les détails de l'infrastructure sous-jacente. Les applications consomment des services de plate-forme (calcul, stockage, réseau) sans connaître les détails du cloud provider.

Trois niveaux d'abstraction courants :

- **Niveau 1 — Aucune abstraction** — les applications connaissent l'infrastructure (connexion directe à RDS, à S3, à un load balancer spécifique)
- **Niveau 2 — Abstraction partielle** — les applications utilisent des API génériques (calcul, stockage, base de données) mappées vers une implémentation spécifique
- **Niveau 3 — Abstraction totale** — les applications utilisent des services de plate-forme (Kubernetes, service mesh, PaaS) qui sont eux-mêmes portables entre clouds

Pour Mapify, Antoine a constaté que les applications étaient au **Niveau 1** : elles connaissaient AWS par leur code (variables d'environnement, SDK spécifiques). La migration vers un autre cloud provider aurait nécessité de réécrire chaque application. Le chantier d'abstraction vers le Niveau 2 (via Terraform et des API génériques) a été lancé sur 18 mois.

***Et toi ?** Ton organisation est-elle au Niveau 1, 2 ou 3 d'abstraction ? Combien de temps faudrait-il pour migrer vers un autre cloud provider aujourd'hui ?*

Troisième principe — Le middleware est un choix structurant, pas une commodité

Le middleware (bus de messages, ESB, API gateway, orchestrateur) structure la circulation de l'information dans le SI. Le choisir sans vision d'ensemble produit des redondances, des conflits, et des coûts cachés.

Le middleware est souvent vu comme une couche technique « sous le capot ». En réalité, c'est une couche structurante qui conditionne la manière dont les applications communiquent. Trois questions doivent être posées avant tout choix de middleware :

- **Le middleware standardise-t-il ou contraint-il ?** Un bus de messages standardise le format des échanges ; un ESB contraint les applications à un modèle spécifique.
- **Le middleware est-il une cible de lock-in ?** Kafka, RabbitMQ, ServiceMix, Mule — chacun verrouille différemment l'organisation.
- **Le middleware permet-il l'observabilité ?** Sans tracing distribué, sans monitoring des flux, le middleware devient une boîte noire.

Pour Mapify, Antoine a constaté que trois middlewares cohabitaient : un RabbitMQ pour les flux asynchrones, un Kong API Gateway pour les APIs REST, un Sidekiq pour les jobs en arrière-plan. Trois technologies, trois modèles de supervision, trois compétences distinctes dans l'équipe. La rationalisation vers un duo (Kafka + Kong) a permis de réduire la complexité opérationnelle de moitié.

***Et toi ?** Combien de middlewares cohabitent dans ton organisation ? Sont-ils gouvernés par une équipe transverse ou par chaque application ? Combien de compétences différentes cela exige-t-il ?*

Quatrième principe — L'observabilité n'est pas un luxe, c'est une infrastructure critique

On ne peut pas exploiter ce qu'on ne peut pas observer. L'observabilité (logs, métriques, traces) est une infrastructure de première classe, pas une couche ajoutée après coup.

L'observabilité est ce qui permet de répondre à trois questions en production :

- **Que se passe-t-il ?** (métriques)
- **Pourquoi cela se passe-t-il ?** (logs)
- **Comment les composants interagissent-ils ?** (traces)

Sans observabilité, les incidents se résolvent à l'aveugle, par essais-erreurs, en consultant les développeurs qui étaient de garde. C'est coûteux, lent, et épuisant pour les équipes.

La contrefaçon classique consiste à acheter un outil d'observabilité (Datadog, New Relic, Grafana) en croyant que le problème est réglé. L'observabilité n'est pas un outil — c'est une **discipline** : instrumenter le code, centraliser les logs, corréler les traces, définir des SLOs.

Pour Mapify, Antoine a lancé un chantier d'observabilité sur 6 mois : OpenTelemetry pour l'instrumentation, Grafana pour les dashboards, PagerDuty pour l'alerting. Le résultat : temps moyen de résolution d'incident divisé par 3, sommeil des équipes de garde préservé.

***Et toi ?** Combien de temps faut-il à ton équipe pour diagnostiquer un incident en production ? Combien d'outils d'observabilité cohabitent ? As-tu des SLOs définis pour les services critiques ?*

Cinquième principe — La sécurité est une propriété architecturale, pas une couche

La sécurité ne s'ajoute pas à la fin — elle se conçoit dès l'architecture. Une architecture qui ne pense pas la sécurité dès le départ produit des systèmes qu'on doit « sécuriser » après coup, généralement avec des résultats médiocres.

L'Open Group identifie cinq domaines de sécurité à traiter en architecture :

- **Identité et authentification** — qui est l'utilisateur ? comment prouve-t-il son identité ?
- **Autorisation** — que peut-il faire ? selon quelles règles ?
- **Chiffrement** — quelles données sont chiffrées, au repos et, en transit ?
- **Audit** — qui a fait quoi, quand ?
- **Conformité** — RGPD, PCI-DSS, ISO 27001 — quelles contraintes s'appliquent ?

La contrefaçon classique consiste à ajouter la sécurité comme une couche (firewall, antivirus, SIEM) sans revoir l'architecture. Cette approche produit des systèmes où la sécurité est contournée par des pratiques officieuses — partage de credentials, services non documentés, accès administrateur trop larges.

Pour Mapify, Antoine a institué un **Security by Design** : chaque nouvelle application passe par une revue d'architecture sécurité avant d'être lancée. Le coût initial est plus élevé (deux à quatre jours par application), mais le coût total est inférieur (pas de remédiation après incident).

***Et toi ?** Combien de revues d'architecture sécurité sont conduites chaque année dans ton organisation ? Combien d'applications sont concernées par chaque revue ? Quel est le délai moyen entre une vulnérabilité détectée et sa remédiation ?*

Application — Mon TRM

À produire séance tenante. Sept sections précises.

1. **Services applicatifs** — Liste des technologies utilisées (langages, frameworks, runtimes) avec criticité.
2. **Services de données** — Liste des SGBD, data warehouses, data lakes avec criticité.
3. **Services d'infrastructure** — Cloud provider, on-premise, virtualisation, conteneurs avec criticité.
4. **Services de plate-forme** — Middleware, orchestrateurs, service mesh avec criticité.
5. **Services de sécurité** — IdP, IAM, secrets management, pare-feu avec criticité.

6. **Services d'observabilité** — Logs, métriques, traces, alerting avec criticité.
7. **Services de déploiement** — CI/CD, IaC, automatisation avec criticité.

Ce TRM est ton premier livrable d'architecture technique. Il décrit l'existant avec ses forces et ses fragilités.

Encadré Au lecteur / À l'animateur

Question d'ouverture : « Décris en 5 minutes la stack technique de ton organisation. Combien de temps as-tu mis pour la documenter ? »

Profils à observer :

- **Le théoricien du cloud.** Décrit la stack cible, pas la stack existante. *Question :* « Combien de services en production ne correspondent pas à ta description ? »
- **L'empileur de solutions.** A ajouté trois middlewares pour répondre à trois problèmes ponctuels. *Question :* « Combien de technologies distinctes pour le même usage ? »
- **Le minimaliste qui n'instrumente rien.** Produit du code sans observabilité. *Question :* « Combien de temps pour diagnostiquer un incident en production ? »
- **Le sécurité-par-feu.** Ajoute des couches de sécurité après coup, sans revoir l'architecture. *Question :* « Combien d'applications échouent à un test d'intrusion ? »

Pièges d'animation :

- Ne pas laisser la discussion dériver vers les éditeurs (AWS vs Azure vs GCP). Rappeler que l'architecture technique est d'abord une question de principes, pas de produits.
- Ne pas confondre TRM et catalogue de produits. Le TRM est un modèle de référence ; le catalogue liste les produits utilisés.

Cas pratique — Architecture d'un jumeau numérique pour la maintenance prédictive (Aéronautique)

Contexte sectoriel : constructeur aéronautique européen, 12 000 employés, 45 lignes d'assemblage. La production est un actif critique : chaque ligne immobilisée représente une perte d'exploitation directe et un risque de retard de livraison aux compagnies aériennes clientes.

Diagnostic de l'existant. Sur les 45 lignes d'assemblage, **1 500 capteurs IoT** génèrent un flux massif de données (vibration, température, pression, acoustique) à une fréquence de 100 Hz. L'enjeu stratégique est de **réduire le taux de pannes sous les 3 %** pour éviter les **1,2 M€ de pertes d'exploitation annuelles** par chaîne arrêtée. Aujourd'hui, la maintenance est corrective ou planifiée à intervalles fixes, sans corrélation avec l'état réel des équipements.

Parties prenantes et écueil politique majeur.

Un point de friction existe entre deux populations :

- Le **Directeur Industriel** (focus ROI) — veut des résultats rapides, mesurables en pertes évitées. Il accepte la complexité technique tant qu'elle produit des gains financiers.
- L'équipe **IoT/Data** (focus complexité technique) — passionnée par les architectures élégantes (Kafka, time-series databases, machine learning), risque de privilégier la sophistication sur la valeur métier.

L'architecte doit **utiliser ArchiMate Motivation Extension** pour lier explicitement les Drivers (Coût des pannes) aux Goals (Maintenance Prédictive) aux Requirements (latence, fréquence d'échantillonnage). Cette traçabilité rend les arbitrages techniques audibles au Directeur Industriel.

Convergence IT/OT (Operational Technology).

L'architecture doit articuler deux couches historiquement disjointes :

- **Couche OT (Operational Technology)** : capteurs physiques, automates SCADA, bus de terrain industriels, contraintes temps réel strictes (latence < 10 ms).
- **Couche IT (Information Technology)** : Message Broker (Kafka), Cloud Data Lake, moteurs d'analyse, machine learning.

Le jumeau numérique est la **passerelle** entre les deux : il reçoit les événements OT, les historise, les modélise, et alimente les algorithmes de prédiction.

Décisions architecturales structurantes :

1. **Edge Computing** : les algorithmes de détection d'anomalie doivent tourner au plus près des capteurs (latence critique). On ne peut pas se permettre un aller-retour Cloud pour chaque échantillon à 100 Hz.
2. **Stockage time-series** : choisir une base optimisée (InfluxDB, TimescaleDB, OpenTSDB) plutôt qu'une base relationnelle classique. La volumétrie (1 500 capteurs × 100 Hz × 24h × 365j) exclut tout autre choix.
3. **Modèle de jumeau numérique** : chaque équipement critique est représenté par un agent logiciel qui agrège son état, son historique, et ses prédictions. Le modèle est exposé via API aux outils de maintenance.

Roadmap incrémentale (pas de Big Bang) :

- **Vague 1 (3 mois)** : instrumentation complète d'une ligne pilote, premiers modèles prédictifs simples (seuils statistiques).
- **Vague 2 (6 mois)** : déploiement sur 5 lignes, modèles de machine learning supervisés.
- **Vague 3 (12 mois)** : généralisation aux 45 lignes, enrichissement par maintenance prescriptive (recommandations d'action).

Livrables attendus de l'apprenant (schéma ArchiMate hybride) :

- Représentation sur trois couches : **Business** (capacité « Maintenance Prédictive »), **Application** (services d'analyse, jumeau numérique), **Technology** (capteurs, broker, base time-series, edge).
- Cartographie des flux temps réel (capteurs → broker → jumeau → algorithmes → tableaux de bord opérationnels).
- Cartographie des risques : obsolescence des automates, vulnérabilités cyber (un automate non patché devient un vecteur d'attaque majeur), latence réseau.

Pièges classiques à éviter :

- Oublier la **latence réseau** : sans Edge Computing, les modèles temps réel ne fonctionnent pas.
- **Vulnérabilité des automates industriels** non patchés face aux cybermenaces (rançongiciels ciblant l'OT en 2024-2025 : incidents documentés sur des sites manufacturiers européens).
- Chercher à tout faire en un projet : la roadmap doit être **incrémentale**, ligne par ligne, pour éviter tout arrêt de production.

Critères d'évaluation : le livrable est validé si l'apprenant (1) sépare explicitement couches IT et OT, (2) justifie le choix Edge Computing par une contrainte de latence chiffrée, (3) propose une roadmap qui n'arrête jamais la production.

Ce que la leçon 10 vous demande

Ouvrez votre dossier d'architecte à la page suivante. Préparez :

- **Un TRM complet** documentant l'existant (sept catégories, criticité notée).
- **Un diagnostic** des composants à risque (obsolescence, dette, complexité).
- **Trois chantiers techniques** à 6 mois : abstraction, observabilité, sécurité by design.

Sources de cette leçon : TOGAF Series Guide to the Technical Reference Model (G229) · Charley Richard, *Mastering Cloud Migration* · Cas Mapify (anonymisé)

Leçon suivante : **L10 — Opportunités et Solutions (Phase E) : la cible priorisée.**

Quiz — 18 questions pour valider votre compréhension

Trois series de six questions. Prenez un crayon, écrivez vos reponses dans l'espace prevu. Score indicatif : 14/18 minimum pour valider la leçon.

A — COMPREHENSION

A1 Qu'est-ce que le TRM (Technical Reference Model) selon l'Open Group ? Pourquoi décrit-il l'existant plutôt que la cible idéale ?

A2 Quels sont les trois niveaux d'abstraction de plate-forme (aucune, partielle, totale) ? En quoi le Niveau 2 ou 3 facilite-t-il la migration ?

A3 Pourquoi le middleware est-il un choix structurant (pas une commodité) ? Quelles sont les trois questions à poser avant tout choix de middleware ?

A4 Qu'est-ce que l'observabilité ? Quelles sont les trois questions auxquelles elle permet de répondre en production (métriques, logs, traces) ?

A5 Quels sont les cinq domaines de sécurité à traiter en architecture (identité, autorisation, chiffrement, audit, conformité) ?

A6 Pourquoi la sécurité doit-elle être conçue dès l'architecture (Security by Design) plutôt qu'ajoutée comme une couche après coup ?

B — AUTO-EVALUATION

B1 As-tu un TRM documenté pour ton organisation ? Décrit-il l'existant ou la cible ? Quelle est la dernière date de mise à jour ?

B2 Ton organisation est-elle au Niveau 1, 2 ou 3 d'abstraction ? Combien de temps faudrait-il pour migrer vers un autre cloud provider aujourd'hui ?

B3 Combien de middlewares coexistent dans ton organisation ? Sont-ils gouvernés par une équipe transverse ou par chaque application ? Combien de compétences différentes cela exige-t-il ?

B4 Combien de temps faut-il à ton équipe pour diagnostiquer un incident en production ? Combien d'outils d'observabilité coexistent ? As-tu des SLOs définis pour les services critiques ?

B5 Combien de revues d'architecture sécurité sont conduites chaque année dans ton organisation ? Combien d'applications sont concernées par chaque revue ?

B6 Décris en 5 minutes la stack technique de ton organisation. Combien de temps as-tu mis pour la documenter ?

C — ENGAGEMENTS A 30/45/60 JOURS

C1 Produire un TRM documentant l'existant sur 7 catégories (applicatifs, données, infrastructure, plate-forme, sécurité, observabilité, déploiement) avec criticité. (Échéance : J+30.)

C2 Diagnostiquer les composants à risque (obsolescence, dette, complexité). (Échéance : J+45.)

C3 Lancer un chantier d'abstraction de plate-forme (Niveau 1 → 2 ou 3) sur 6 mois. (Échéance : J+180.)

C4 Instrumenter 3 services critiques avec OpenTelemetry, Grafana, et PagerDuty. (Échéance : J+60.)

C5 Définir 3 SLOs (Service Level Objectives) pour les services critiques et configurer les alertes. (Échéance : J+45.)

C6 Lancer un programme Security by Design : revue d'architecture sécurité systématique pour toute nouvelle application. (Échéance : J+60.)
