Ingénierie des Architectures Logicielles · fiche contrôle

B. Vella (base N. Ferry, G. Rocher) · cours 1 à 7, hors SOAP/REST · exemples : notre projet e-santé

Comment lire la fiche. Le contrôle est une restitution rédigée (le cours 7 écarte le QCM). Chaque carte donne la définition, ce qu'elle apporte, sa limite, et un exemple tiré du projet. L'étiquette noire **É3** renvoie à l'étape du plan de dissertation (dernière section) où la notion sert d'argument.

É1 Besoin personas, données, US, priorités	**É2 Domaine** DDD stratégique puis tactique	**É3 Qualités** attributs, contraintes, compromis	**É4 Style** couches, pipeline, EDA, services	**É5 Données et échanges** sync/async, garanties, cohérence
É6 Décrire, décider C4, vues, ADR	**É7 Risques conception** FMEA, Bowtie	**É8 Risques prod** fallacies, résilience	**É9 Détecter** observabilité, tests de contrat	**É10 Durer** modernisation, déploiements

↔ **Idée centrale** une architecture est une suite de **compromis** justifiés par le métier et les contraintes. Le prof le répète du cours 1 au cours 7 : toute réponse doit dire ce qu'on gagne, ce qu'on paie, et pourquoi ça vaut le coup ici.

1 Ce qu'est une architecture

cours 1

Définition et préoccupations

Johnson cité par Fowler, *Who needs an architect?* **É1** **É3**

Définition : la **compréhension partagée** par les développeurs experts de la façon dont le système est découpé en **composants** et dont ils interagissent par des **interfaces**. Seuls les éléments structurants, compris de tous, en font partie.

Propriétés : répond aux besoins utilisateurs · peut suivre un style · **n'est pas figée** · influence l'organisation de l'équipe et inversement · décrit le travail à faire · facilite la communication, la réutilisation, le repérage de ce qui va varier.

Famille	Contenu
Besoins fonctionnels	implémentation, performances
Attributs de qualité	modularité, évolutivité, disponibilité, sécurité, passage à l'échelle
Contraintes	déploiement et infra, compétences de l'équipe, budget

Ces attributs sont **liés et parfois contradictoires** → **architecture = compromis**.

! Piège l'architecture n'est pas un diagramme : c'est l'accord de l'équipe. Un diagramme faux qui « fait joli » ne vaut rien (voir as-built vs as-designed, ch. 7).

Projet contrainte d'équipe : 3 personnes, un semestre, Docker seul chez le correcteur → monolithe modulaire pour le backend plutôt que 12 microservices.

Modularité · Separation of Concerns

cohésion, couplage, Interceptor **É4**

Diviser un objet complexe en modules **développables indépendamment**. Objectif : **forte cohésion** (ce qui va ensemble reste ensemble) et **faible couplage** (peu de flèches entre modules).

Réutilisabilité **non garantie** : il faut des tests solides, et c'est dur quand métier et technique sont mélangés dans le code. Les préoccupations **transverses** (sécurité, logs) s'isolent avec l'**Interceptor pattern** : un filtre appliqué à toutes les requêtes, sans que le code métier le sache.

Projet le journal d'accès (audit) est un filtre de l'API qui inscrit chaque lecture d'un patient : préoccupation transverse isolée, les contextes métier l'ignorent.

2 Pipeline, stream, lambda, couches, tiers

cours 2 · TP1

Architecture en pipeline et traitement de flux

source, filtre, canal **É4** **É5**

Pour un **flux de données similaires** (capteurs) : **source** (produit) → **filtres** (traitements) reliés par des **canaux** (souvent unidirectionnels) ; le **pipeline** orchestre le tout.

Notion	Les deux options
Données	**bounded** (fin connue, fichier) / **unbounded** (capteurs)
Traitement	**batch** (ensemble fini, logs d'hier) / **stream** (à l'arrivée)
Fenêtre	**fixe** (toutes les 5 min sur 5 min) / **glissante** (en continu)
État	**stateless** (filtrer < 0) / **stateful** (moyenne, tendance)
Partition	**shuffle** (équilibré) / **fields** (même clé → même worker)
Perf	**latence** (délai par donnée) / **débit** (données par seconde)

Risques : désordre, échec d'un traitement, perte, double traitement → **quelles garanties veut-on ?**

Apache Storm : **topology** (graphe complet), **spout** (source : Kafka, API, capteur), **bolt** (étape), **grouping** (partitionnement), **stream** (suite de tuples).

TP1 séismes USGS : QuakeSpout → CsvSplitBolt (shuffle) → FormatBolt (fields « place »). Même chose en Node-RED : inject → http → csv → function → worldmap.

Lambda architecture : un système réel combine plusieurs styles. Ex. une couche batch (recalcul exact sur tout l'historique), une couche speed (stream, résultat approché immédiat), une couche serving qui fusionne.

! Piège le fields grouping garantit qu'une clé reste chez un worker (donc ordre et état locaux), pas un bon équilibrage : une clé très active surcharge un worker.

Projet fréquence cardiaque = stream unbounded, **stateful**, fenêtre **glissante de 30 s** (tachycardie, bradycardie, perte de signal) · Kafka clé `patientId` = fields grouping : les mesures d'un patient restent ordonnées dans une partition.

Couches (layers) et tiers

type d'architecture le plus courant É4

Couches fonctionnelles : **présentation · métier · persistance · base**. Chaque couche peut englober plusieurs composants, forme une abstraction, ne parle qu'à ses voisines.

Fermée : accès aux seules couches adjacentes · **ouverte** : la couche peut être contournée. Transverses (sécurité, monitoring) : présentes dans plusieurs couches.

Sinkhole anti-pattern : des couches qui ne font que relayer la requête.

Tier = unité de **déploiement** séparée, avec du **réseau** entre tiers · **layer** = séparation **logique**. Chaque tier est serveur pour celui du dessus, client pour celui du dessous.

n	Découpage
2	client · serveur
3	client · métier · base
4	client · serveur front · métier · base
5	client · front · API gateway · métier · base

+ Forces séparation et abstraction : tests, évolution et maintenance simples · cohésion technique

– Limites couches inutiles (sinkhole) · grosse granularité, passage à l'échelle difficile · en distribué, coût de chaque saut · **dispersion métier** : une fonctionnalité touche toutes les couches

3 Event-Driven Architecture

cours 3 · TP2 MQTT

Pourquoi des événements · pub/sub

découplage É4 É5

Dans un pipeline, l'émetteur doit connaître le canal, la suite du traitement, et chaque nouvelle équipe intéressée → **couplage fort et dépendance organisationnelle**.

En EDA, le producteur **annonce qu'il s'est passé quelque chose** (événement) ; les intéressés s'abonnent et réagissent (**Publisher-Subscriber**). Composants qui ne se connaissent pas (**space decoupling**) et n'ont pas à être présents en même temps (**time decoupling**).

Topologie	Principe
Mediator	un orchestrateur dédié déclenche les étapes : flux explicite, point central
Broker	chaque processeur traite puis émet l'événement suivant : chorégraphie, pas de vue globale

+ Forces découplage technique et organisationnel · ajout de consommateurs sans toucher au producteur · parallélisme · résiste à l'absence d'un consommateur · déploiements indépendants

– Limites asynchronisme complexe · doublons, retries, idempotence · cohérence à garantir · workflow dur à déboguer · **contrat d'événement** à gérer

Garanties de livraison, ordre, ACK

É5 É8

Garantie	Mécanisme	Coût
at most once	publié une fois	perte si consommateur absent
at least once	renvoi jusqu'à ACK	doublons → consommateurs **idempotents**
exactly once	ids uniques, détection de doublons, coordination	cher, rarement vrai de bout en bout

Auto ACK (défaut, dès réception) : prouve la réception, **pas le traitement**. **Client ACK** (manuel, après traitement) : permet de distinguer les erreurs. Selon l'erreur : retry immédiat · retry avec délai · nombre max · **Dead Letter Queue** · émettre un événement d'erreur.

Ordre : stocker les messages (FIFO, LIFO, FEFO ; par id, clé, partition) · plusieurs consommateurs en parallèle cassent l'ordre global → **partitionner** : ordre garanti **dans** une partition.

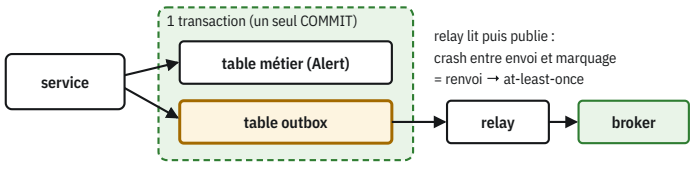
▸ **Projet** FC en **at most once** (QoS 0, lot trop vieux jeté : une valeur toutes les secondes, en perdre une est acceptable) · mesures ponctuelles et alertes en **at least once**, idempotence par `eventId` ou upsert sur (`deviceId`, mesure, date) · alerte non stockable : retries à délai croissant puis `alerts.dlq` + alerte technique à l'astreinte · message illisible : `ingestion.rejected` **sans retry** (erreur non transitoire).

Transactional Outbox

mettre à jour la base ET publier É5 É8

Crash entre l'update DB et l'envoi de l'événement : base d'abord → événement jamais envoyé ; événement d'abord → la base ne le reflète pas. **Incohérent dans les deux cas.**

Solution : écrire la donnée métier et la ligne d'outbox **dans la même transaction** ; un **relay** lit l'outbox et publie. L'enregistrement est atomique ; le relay peut crasher entre l'envoi et le marquage → **at least once**.



▸ **Projet** une alerte et ses notifications d'escalade sont écrites dans la même transaction (`outbox_event`), le relay publie sur `notifications` chaque seconde, et ne publie pas une notification devenue inutile (alerte déjà prise en charge).

Le contrat d'événement · MQTT

ce que producteurs et consommateurs partagent **É5** **É10**

Léger (un id) : petit, mais le consommateur rappelle un service → couplage. **Lourd** (**Event-Carried State Transfer**) : autonome, mais données dupliquées, possiblement obsolètes.

Sémantique : `{"temperature": 39,5}` ne dit ni l'unité, ni ponctuel ou moyen, ni brut ou arrondi → `celsiusTemperature` ou champ `unit`.

Métadonnées : type, id, partition ou corrélation, date, **version** (la donnée n'est pas une métadonnée).

Évolution : 15 consommateurs de `name` dans 6 équipes → versionner ne suffit pas, préférer les changements **backward-compatible** (ajouter un champ, ne pas en renommer).

MQTT : pub/sub léger pour l'IoT et les objets peu puissants. Topics hiérarchiques `pays/region/cp/maison/solarEnergy`. Wildcards : `+` un niveau (`france/paca/+/house42/solarEnergy`), `#` tous les niveaux, en fin (`france/paca/#`). Gère la QoS : 0 at most once, 1 at least once, 2 exactly once.

TP2 Mosquitto local (2.x : `listener 1883` + `allow_anonymous`), Node-RED, monitoring CPU/mémoire multi-machines avec alarmes.

› **Projet** chaque événement porte `eventId`, `eventType`, `schemaVersion`, avec un JSON Schema par type dans `docs/schemas/` · topics de la plateforme `patients/{p}/devices/{d}/telemetry/{mesure}`, le bridge s'abonne avec `+`.

4 Domain-Driven Design

cours 4

DDD stratégique : comprendre et découper le métier

niveau macro **É1** **É2**

Les arbitrages (coût, délai, marché, environnement) ne se font pas au niveau du code : il faut comprendre **besoins, contraintes, risques** du métier. Concevoir à partir de là = DDD.

Outil	Ce que c'est	À quoi ça sert
Ubiquitous Language	vocabulaire commun experts et devs, jusqu'aux noms de classes	modèle mental commun, moins de malentendus
Domain Storytelling	ateliers (experts, PO, designers, devs) ; scénarios simples dessinés	collecter et valider la connaissance
Subdomains	**core** (différenciant) · **supporting** (nécessaire, non différenciant) · **generic** (déjà résolu ailleurs)	**prioriser l'effort** : on soigne le core, on achète ou réutilise le generic
Bounded Context	frontière où un modèle et son langage restent cohérents ; plusieurs par domaine	éviter un gros modèle incohérent
Context Map	relations entre contextes : upstream/downstream, modèle exposé, contrats	voir les dépendances et où protéger son modèle

↔ **Relations de context map** (vocabulaire DDD standard, utile en copie) · **ACL** anti-corruption layer : on traduit le modèle de l'autre · **Published Language** : format d'échange partagé et versionné · **Customer/Supplier** · **Conformist** : on prend le modèle amont tel quel · **Shared Kernel** · **Open Host Service** : API publique documentée

! Pièges **Bounded Context ≠ microservice** : un contexte peut être un module de monolithe, un service ou plusieurs · DDD est agnostique : il donne des **candidats** au découpage, pas une architecture

› **Projet** core : **Surveillance et alertes**, **Mesures vitales** · supporting : activité au domicile, coordination des soins · generic : identité (JWT, TOTP), connectivité IoT (Mosquitto), audit · le mot « chute » change de sens : pic d'accélération pour l'appareil, événement `fall` pour l'ingestion, incident avec escalade pour la surveillance → 3 contextes · **ACL** : un `DeviceAdapter` par modèle de montre traduit le format fabricant vers le format canonique.

DDD tactique : modéliser l'intérieur d'un contexte

niveau micro **É2**

Élément	Définition	Exemple cours	Exemple projet
Entity	identifiable unitairement, l'identité dure	`Device` + deviceID	`Patient`, `Alert`
Value Object	défini par sa valeur, immuable, remplacé	`Temperature(37,5, CELSIUS)`	`VitalThresholds`, `Qualification` (records Java)
Aggregate	frontière de cohérence métier, un seul point d'entrée : la **root**	`PatientMonitor` (capteurs, seuils)	`Alert` : OPEN → TAKEN_IN_CHARGE → CLOSED, sinon 409
Repository	interface pour charger et sauver les agrégats, au niveau métier		upsert idempotent des mesures
Domain Event	fait métier significatif, devient souvent un message Kafka ou MQTT		`FallAlert`, `InactivityAlert`
Domain Service	règle métier qui n'appartient à aucune entité (≠ application service)		`EscalationPlan` : qui prévenir et quand

! Piège **Domain Model ≠ Data Model** : ne pas mettre toute la logique dans des services autour de classes vides (modèle anémique) · pas obligé d'utiliser tous les concepts, seulement ceux qui ont un rôle.

5 Monolithe ou micro-services

cours 5

Que faire des morceaux issus du DDD ?

É4 É5

Monolithe = application **déployée d'un bloc** (pas « tout dans un fichier »). Avec des frontières internes propres : **monolithe modulaire**.

Limites quand : un module change chaque semaine et un autre une fois par an (on redéploie tout) · un seul module est chargé (on duplique tout).

Micro-service : application déployable indépendamment, autonome (réseau), avec une capacité métier cohérente (pas forcément 1 pour 1 avec le DDD).

Principes : responsabilité **métier** unique (pas forcément « micro ») · domaine délimité (bounded context) · **couplage faible obligatoire**, sinon **distributed monolith** · **contrat d'API** versionné.

Avantages : passage à l'échelle et ressources par service · résilience (une panne isolée, redémarrage seul) · équipes et technos indépendantes, seule contrainte le contrat.

Coût : un appel de fonction (réussit ou lève une erreur) devient un appel réseau qui peut réussir, échouer, **ne jamais répondre**, réussir très lentement, réussir mais perdre la réponse, être traité **plusieurs fois**. On échange de la complexité interne contre de la complexité **distribuée**.

Données : base commune = gros couplage · base par service = transactions multi-domaines difficiles → **SAGA** (chorégraphie ou orchestration, avec actions de compensation) → **cohérence éventuelle**.

↔ **Accès** **API Gateway** : façade externe qui reçoit et dispatche, authentifie, limite le débit, agrège (piège : y mettre le métier recrée un monolithe) · **Backend For Frontend** : une gateway par type de client · **Service Discovery** : registre mis à jour à chaque instance créée ou détruite, sert au load balancing. Rien n'est gratuit : il faut découvrir, répartir, automatiser.

▸ **Projet** backend = **monolithe modulaire** (un paquet par contexte, une base) car les contextes changent ensemble et l'équipe est petite ; services séparés **aux bords** (un simulateur par type d'objet, bridge MQTT → Kafka) où l'hétérogénéité et la charge le justifient. Le pull planner se répartit les patients par partition Kafka : on peut lancer plusieurs instances du backend.

6 Micro-services dans la vraie vie

cours 6

Les 8 fallacies of distributed computing

Sun Microsystems, années 90 : hypothèses du dev local É8

Illusion	Réalité en production	Illusion	Réalité
le réseau est fiable	paquets perdus, DNS, LB en panne	la topologie ne change pas	autoscaling, crash, rolling → discovery
latence nulle	variable, s'additionne sur le flow	un seul administrateur	équipes, cloud, tiers
bande passante infinie	congestion, gros payloads, retries	transport gratuit	coût en temps et en argent
réseau sûr	chiffrer, authentifier, autoriser	réseau homogène	protocoles, OS, 4G/Wifi, PC/mobile

Personne ne croit le réseau parfait, mais on **oublie un cas d'erreur**, surtout derrière les abstractions des frameworks. Les bugs rares deviennent **fréquents à grande échelle** : regarder les cas limites, pas la moyenne.

Quand un appel échoue

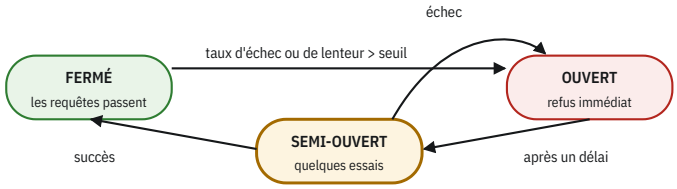
nature → timeout → retry → circuit breaker É8

1. Transitoire ? Retenter : **429**, **5xx** sauf 500 et 520 (503, 504). Inutile : **4xx** sauf 429, **500**, **520**, crash.

2. Timeout (par appel et par flow, calé sur les délais normaux) · trop court : un appel lent devient un faux échec et un retry qui surcharge · trop long : threads, sockets, connexions bloqués.

3. Retry si le service est **idempotent** (**idempotency-key** dans la requête) · surcharge probable : **exponential backoff** + **jitter** (aléa, évite les vagues synchronisées) · savoir s'arrêter : **retry limit** (par requête), **retry budget** (par service et par unité de temps).

4. Circuit breaker si la panne est **avérée** (retenter gaspille) :



Décision sur : taux d'échecs, taux d'appels lents, **fenêtre glissante**, nombre minimal d'appels. Ouvert : protège le service malade et économise le client (**fail fast**). Semi-ouvert : reprise contrôlée, pas 50 000 clients d'un coup. **Ciblé**, jusqu'à un breaker par opération.

Limiter la contamination

côté serveur et côté client É8

Côté	Pattern	Idée
client	**Bulkhead**	ressources compartimentées **par dépendance** : une dépendance lente ne bloque pas tout le service
client	**Graceful fail**	réponse dégradée : **partielle** (sans les recommandations), **ancienne** (cache), **fallback** (service moins précis), et on prévient l'utilisateur
serveur	**Rate limiting**	refus au-delà d'un seuil : **429 + Retry-After**
serveur	**Load shedding**	en surcharge, ne servir que l'essentiel : signaler une anomalie cardiaque avant afficher un avatar

! Piège de calcul requêtes simultanées = débit × durée (loi de Little) · 100 req/s × 10 ms = **1** · 10 req/s × 10 s = **100** (la slide écrit 1000, le calcul donne 100). Limiter en req/s ne protège pas d'appels longs.

▸ **Projet** rejet immédiat des lots de FC trop vieux (on préfère du frais) · un capteur qui ne répond pas après plusieurs demandes passe « injoignable » au lieu d'être relancé sans fin (retry limit).

Observabilité et service mesh

trouver la maladie à partir du symptôme **É9**

Monitoring : grandeurs choisies à surveiller · **observabilité** : capacité à comprendre l'état du système à partir de ses données. Trois signaux :

Logs (événement discret) : centralisés, requêtables (format fixe), rattachables à un flow (id propagé). Piège : trop en faire (un rapport d'étudiant : 3,45 To/an gagnés sur 10 logs).

Métriques (agrégées dans le temps) : **compteur** (croît, on lit son taux), **gauge** (instantané), **histogramme** (distribution). Requêtes : **RED** = Rate, Errors, Duration ; + **Saturation** = 4 golden signals de Google. La moyenne cache la **tail latency** (regarder p99).

Traces : parcours d'une requête, faites de **spans** (début, durée, statut, opération), liées par **context propagation** (traceId, spanId, parentId).

Service mesh : retries, timeouts, sécurité, réseau, observabilité ne sont pas du métier et on ne veut pas les recoder dans chaque service → couche d'infrastructure dédiée.

Data plane (sidecar) : un proxy par service qui accepte, route, répartit, mesure, chiffre, retente. **Control plane** : la configuration qu'il applique (routes, certificats, retry et timeout, load balancing, contrôle d'accès).

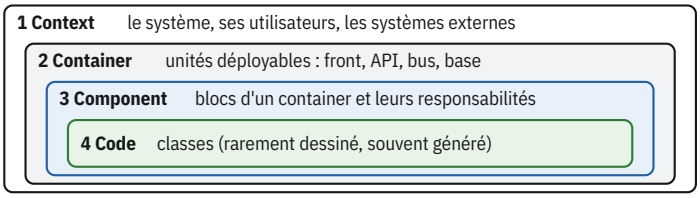
> **Projet** logs JSON, traces et tableaux Grafana (profil d'observabilité) · page de supervision admin : santé des services, retard de l'outbox, nombre d'événements en DLQ · l'astreinte reçoit une alerte technique si un service ne répond plus.

7 Décrire, décider, faire évoluer

cours 1 et 7

Décrire : C4 et vues architecturales

c4model.com **É6**



Boîtes et flèches pour commencer ; UML est un standard mais **pas obligatoire** (autres approches : OASIS, C4).

C4 = zooms successifs sur le même système. Les **vues architecturales** montrent des **aspects différents** : **logique** (sous-systèmes groupés), **déploiement** (infra, dépendances), **implémentation** (composants et responsabilités), runtime, sécurité.

> **! Piège** ne pas confondre un niveau de zoom C4 et une vue · représenter tout jusqu'au code est illisible : choisir les vues utiles au lecteur.

> **Projet** Context : Auprès + senior, aidant, soignant, admin, objets, SMS · Container : Angular, Quarkus, bridge, Mosquitto, Kafka, Postgres, simulateurs.

Décider : Architecture Decision Record

Nygard, enrichi en MADR **É6** **É10**

Titre (la décision) · **Contexte** (problème, besoins, contraintes, date) · **Options envisagées** · **Décision** et pourquoi · **Conséquences** (avantages, inconvénients, risques) · **Statut** : Proposed, Accepted, Deprecated, Superseded.

Bonnes pratiques : court et clair · pas pour chaque micro-décision · écrit au moment de la décision · **garder les alternatives rejetées** · versionné.

Exemple projet. *ADR : Kafka derrière MQTT.* Contexte : objets peu puissants, plusieurs consommateurs internes, rejouer un flux, ordre par patient. Options : MQTT seul, RabbitMQ, Kafka. Décision : MQTT pour les objets (léger, QoS, Last Will), Kafka en interne (partitions par patient, rétention, consumer groups). Conséquences : + rejouable, scalable par partition ; – un bridge et un broker de plus à opérer. Statut : Accepted (RabbitMQ abandonné).

Moderniser une architecture existante

cas le plus fréquent pour un ingénieur **É10**

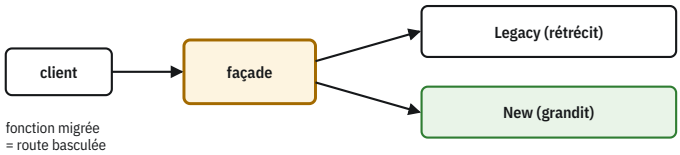
Migrer = remplacer une techno par une autre · **moderniser** = améliorer l'adéquation du système à ses besoins et contraintes (pas forcément tout réécrire : retirer une dépendance, isoler un module).

Quand ? Pas l'âge (vieux = équipe formée, CI rodée, besoins satisfaits). Quand il y a **inadéquation entre capacités et besoins** actuels ou futurs : dépendances non maintenues, maintenance hors de prix, compétences rares. Une migration coûte cher, on ne la fait pas pour rien.

Architecture archeology : on veut le système **as-built**, pas as-designed (la requête SQL directe non documentée d'il y a 7 ans). Analyse **métier** (spec, experts), **statique** (code, dépendances, requêtes DB, CI/CD, tests), **dynamique** (logs, traces, trafic).

Par blocs : 1 partir des **problèmes**, pas des solutions (« pas de micro-services » n'est pas un problème ; « l'équipe ne peut pas déployer seule » en est un) · 2 **hotspots** : dette × forte activité (historique git, incidents, **change coupling**) · 3 **facteur limitant** · 4 **prioriser** coût, gain, risque, **cost of delay**.

Strangler Fig : façade devant l'existant, qui bascule fonction par fonction vers le nouveau système (évite le big bang : pas de retour client, risque de régression, bascule).



Données	Prix
à froid : arrêt écritures, export, import, reprise	interruption, vérifier l'import
à chaud : snapshot + **Change Data Capture**, rattrapage au cutover	complexe, interruption très courte
dual write : écrire dans les deux	double travail, **divergence** si une écriture échoue

API : **Expand-Migrate-Contract** (v1 et v2 en parallèle, délai annoncé, puis retrait de v1) · tests **provider-driven** (l'implémentation respecte le contrat) et **consumer-driven** (les besoins des consommateurs restent servis), valables aussi en EDA.

Déploiement : **rolling** (instances remplacées peu à peu) · **canary** (5, 20, 50, 100 % du trafic) · **dark launch** derrière un feature flag + **shadow traffic** (parallel run, on compare). Prix : plus d'instances, de coûts, de monitoring ; rollback pas toujours possible.

8 Gestion des risques : FMEA et Bowtie

cours 1

Démarche

É7

Anticiper sans connaître le futur : décisions sur une connaissance partielle, ce qui n'empêche pas d'agir. **Réactif** (gestion de crise) vs **proactif** : identifier, évaluer les impacts, préparer la mitigation. **Évaluation** : faiblesses, impacts, priorité ≈ **P(risque) × impact**. **Contrôle** : plan de mitigation, mise en œuvre, **monitoring**. Méthodes : FMEA, CORAS, LINDDUN (vie privée), STRIDE (menaces sécurité), Bowtie.

FMEA · Failure Mode and Effects Analysis

élément par élément, chiffré É7

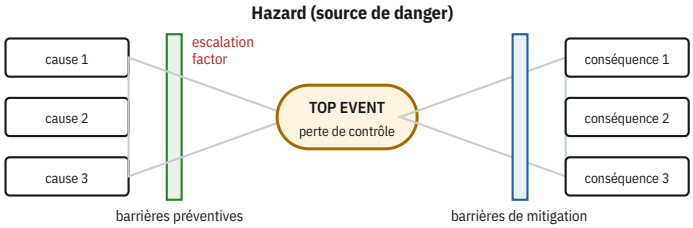
Pour chaque élément : comment il casse (**mode**), conséquences (**effet**), gravité (**SEV**), causes, probabilité (**OCC**), contrôle existant, détectabilité (**DET**, élevée = **mal** détecté), priorité **RPN = SEV × OCC × DET** (notes 1 à 10), action. Agir sur les RPN hauts **et** sur toute gravité 9 ou 10.

Élément	Mode	Effet	S	Cause	O	Contrôle	D	RPN	Action
chaîne d'alerte chute	alerte non enregistrée	personne au sol sans aide	10	base indisponible	2	retries, DLQ, alerte astreinte	2	40	gravité 10 : garder la redondance médaillon
montre (FC)	mesures perdues en coupure	trou dans le suivi	4	Wifi, QoS 0	6	aucun	7	168	QoS 1 + file locale (US-03)

Lecture : la ligne 1 a un RPN bas mais une gravité maximale, elle reste prioritaire. C'est ce raisonnement que le prof attend, pas 40 lignes à RPN ridicule.

Bowtie

un événement redouté, ses causes et ses suites É7



Hazard : ce qui est dangereux par nature · **threats** (causes) · **barrières préventives** · **top event** (perte de contrôle) · **barrières de mitigation** · **conséquences** · **escalation factors** : pourquoi une barrière lâcherait (on peut leur ajouter des contrôles).

Cours (porte) : perte réseau de la caméra → barrière : reconnaissance faciale locale (escalade : crash du logiciel local) → top event : contrôle d'accès indisponible → mitigation : clé physique → le propriétaire entre quand même.

Projet hazard : surveiller seule une personne âgée · cause : montre déconnectée (batterie, Wifi) · prévention : heartbeat + Last Will → alerte « appareil hors ligne », batterie faible signalée · escalade : le broker lui-même tombe · top event : chute non détectée · mitigation : bouton SOS du médaillon, inactivité anormale vue par les capteurs de présence, passage de l'infirmière.

↔ **FMEA ou Bowtie** (réponse à l'exercice du cours) · FMEA : systématique, chiffrée, priorise, couvre tout un système connu ; lourde, ne montre pas les chaînes de causes ni les barrières · Bowtie : visuel, centré sur quelques événements graves, sépare prévenir et limiter, montre où une barrière lâche ; pas exhaustif, pas de score · FMEA pour inventorier composant par composant, Bowtie pour les événements critiques (sécurité des personnes) et pour communiquer.

DS Répondre à la question du contrôle

sujet donné au cours 7

Sujet

« Comment passer d'un **besoin métier** à une architecture logicielle **cohérente**, **adaptée aux contraintes** du projet et **maintenable à long terme** ? Vous détaillerez les **différentes étapes**, en **justifiant leur pertinence** et en détaillant les **éléments à prendre en compte** pour chacune. Vous discuterez également des moyens de **détecter** et **limiter** les risques architecturaux. »

Ce que chaque mot du sujet appelle

à faire au brouillon en 5 min

Mot	Ce que le correcteur veut voir
besoin métier	analyse du besoin (personas, données, US priorisées) et DDD : langage, domain stories, sous-domaines, contextes
cohérente	forte cohésion, faible couplage, découpage aligné sur les bounded contexts, contrats clairs
adaptée aux contraintes	attributs de qualité priorisés, contraintes (équipe, budget, infra), **compromis assumés**, pas de micro-services par défaut
maintenable à long terme	modularité, contrats versionnés et backward-compatible, ADR, observabilité, évolution progressive
étapes · pertinence · éléments	pour **chaque** étape : quoi, pourquoi maintenant, à quoi faire attention
détecter · limiter les risques	**deux verbes, deux listes** : FMEA, Bowtie, fallacies, observabilité, archéologie / patterns de résilience, idempotence, déploiement progressif, strangler

Plan conseillé : 3 parties qui suivent la vie du projet

chaque étape : quoi · pourquoi · points d'attention

Introduction (≈ 8 lignes) : définition (compréhension partagée des composants et interfaces, Johnson) · une architecture répond à des besoins fonctionnels, des attributs de qualité et des contraintes souvent contradictoires, donc c'est un **compromis** · problématique : comment rendre ces compromis **justifiés par le métier** et **réversibles** ? · annonce du plan · fil rouge annoncé : un exemple (notre suivi de seniors à domicile).

Étape	Quoi faire	Pourquoi (pertinence)	Points d'attention
I. Comprendre le métier avant de dessiner			
É1 Besoin	objectifs, personas, données d'entrée (capteurs, saisies), vues de sortie, US en MoSCoW	l'architecture répond aux besoins utilisateurs, pas à une techno	quels besoins sont critiques (alerte chute) vs confort
É2 Domaine	ubiquitous language, domain stories avec les experts, sous-domaines core/supporting/generic, bounded contexts, context map	vocabulaire commun ; priorise l'effort sur le core ; donne des candidats au découpage	BC ≠ microservice ; un mot qui change de sens = deux contextes
É3 Qualités et contraintes	lister et **classer** disponibilité, sécurité, scalabilité, évolutivité, performance ; contraintes équipe, budget, infra, délai	c'est ce classement qui départage deux architectures	ils se contredisent : expliciter le compromis choisi
II. Concevoir et justifier			
É4 Style	choisir le ou les styles : couches, pipeline/stream pour les flux capteurs, EDA pour découpler, monolithe modulaire ou micro-services	chaque style répond à un besoin (Lambda : on les combine)	micro-services = complexité distribuée ; sinkhole ; distributed monolith
É5 Données et échanges	sync ou async, garanties (at most / at least once), idempotence, ordre par partition, contrat d'événement, stockage, cohérence (outbox, SAGA)	la cohérence du système se joue aux frontières	doublons, désordre, perte ; évolutions backward-compatible
É6 Décrire et décider	C4 (context, container, component), vues utiles (déploiement, sécurité), ADR pour les choix structurants	compréhension partagée ; on saura dans 2 ans **pourquoi**	ADR courts, alternatives rejetées gardées, versionnés
III. Détecter et limiter les risques, faire durer			
É7 Détecter à la conception	FMEA (RPN = S × O × D) ou Bowtie sur les événements graves	proactif plutôt que gestion de crise	peu de lignes pertinentes ; gravité 9-10 traitée même à RPN bas
É8 Limiter en production	prendre au sérieux les 8 fallacies : timeout, retry + backoff + jitter, circuit breaker, bulkhead, rate limiting, load shedding, graceful fail, DLQ	en distribué tout appel peut échouer, lentement ou deux fois	retry seulement si transitoire et idempotent
É9 Détecter en production	logs, métriques (RED + saturation), traces ; tests provider et consumer-driven ; CI/CD avec tests bloquants	on voit le symptôme, il faut remonter à la cause	trop de logs ; moyenne vs tail latency
É10 Faire évoluer	archéologie as-built, hotspots, strangler fig, migration de données, expand-migrate-contract, rolling, canary, dark launch	une architecture n'est pas figée ; moderniser sur une inadéquation, pas sur l'âge	partir des problèmes ; coût de l'inaction ; rollback

Conclusion (≈ 6 lignes) : le passage du besoin à l'architecture est une **boucle**, pas une séquence : le métier fixe les priorités, les compromis sont écrits (ADR), les risques sont analysés puis surveillés, et l'architecture évolue par petites étapes réversibles. Ouverture : la plupart du temps on hérite d'une architecture, et les mêmes outils (DDD, ADR, observabilité) servent à la moderniser.

Phrases qui rapportent

« Ce choix sert l'attribut X, au prix de Y ; ici X est prioritaire parce que… »

« Un bounded context n'impose pas un micro-service : nous avons gardé un monolithe modulaire parce que… »

« Kafka livrant au moins une fois, les consommateurs sont idempotents grâce à l'identifiant d'événement. »

« Le risque est détecté par… et limité par… » (toujours les deux verbes).

« Cette décision est tracée dans un ADR, avec les alternatives rejetées. »

Ce qui fait perdre des points

Réciter les notions sans dire **pourquoi** ni **dans quel cas** · présenter les micro-services comme la bonne réponse · aucune limite ni alternative · oublier la partie risques ou ne traiter que « détecter » · aucun exemple concret · liste à puces sans liaison entre les étapes · confondre layer et tier, C4 et vues, BC et service.

Temps (durée non annoncée) : environ 15 % plan au brouillon, 75 % rédaction, 10 % relecture. Un schéma simple (fil des étapes, bowtie) se recopie vite et structure la copie.

Sources : Cours_01 à Cours_07 de B. Vella (2026-2027), TP1 Storm et Node-RED, TP2 MQTT, documentation du projet d'équipe (DDD, événements, README). Hors programme ici : cours SOAP, REST, CoAP. Relations de context map nommées et loi de Little : compléments standards, non détaillés sur les slides.