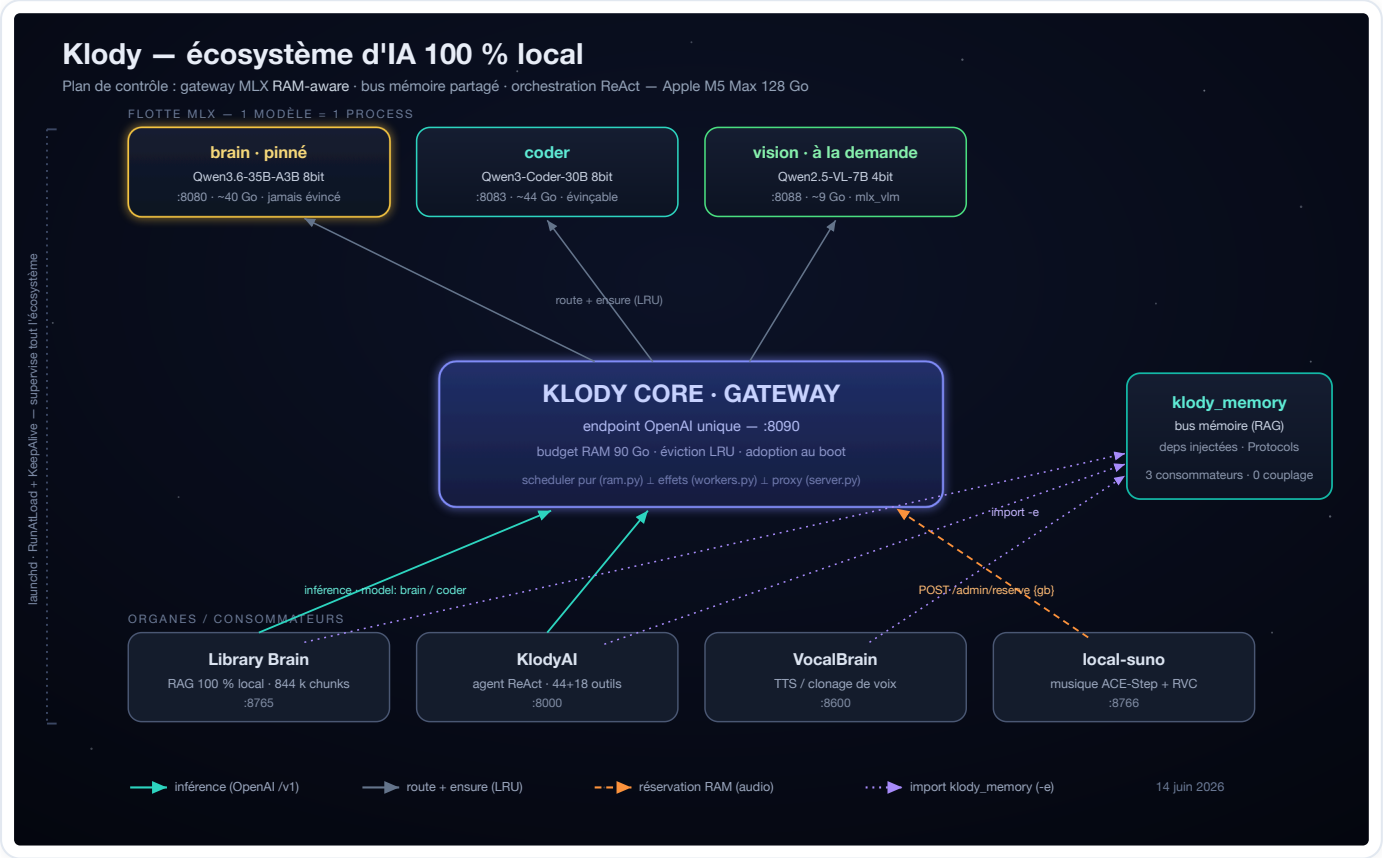
Klody — Architecture d'un écosystème d'IA 100 % local

> **Portfolio technique** — conception, méthodes, pratiques et stack. Auteur : **klodynlov** · Machine : Apple **M5 Max, 128 Go** · macOS (arm64) · 100 % local, zéro cloud, zéro télémétrie. Période couverte : **9 mai → 14 juin 2026** · Document généré le **14 juin 2026**.



0. Résumé exécutif

En cinq semaines, j'ai conçu et mis en production un **écosystème d'intelligence artificielle entièrement local** : cinq applications spécialisées (RAG documentaire, agent de code, synthèse vocale, génération musicale, dashboard desktop) puis — surtout — le **plan de contrôle** qui les unifie : **Klody Core**.

Le cœur du travail n'est pas « encore une app IA », mais une **infrastructure** : un *gateway* d'inférence partagé, conscient de la mémoire vive, qui mutualise des modèles de plusieurs dizaines de gigaoctets entre toutes les applications, sous un budget RAM unique avec éviction intelligente. Autour de lui, un **bus mémoire** réutilisable (moteur RAG extrait en package à dépendances injectées) et un **agent orchestrateur** (boucle ReAct) qui pilote les organes.

Ce que ce projet démontre :

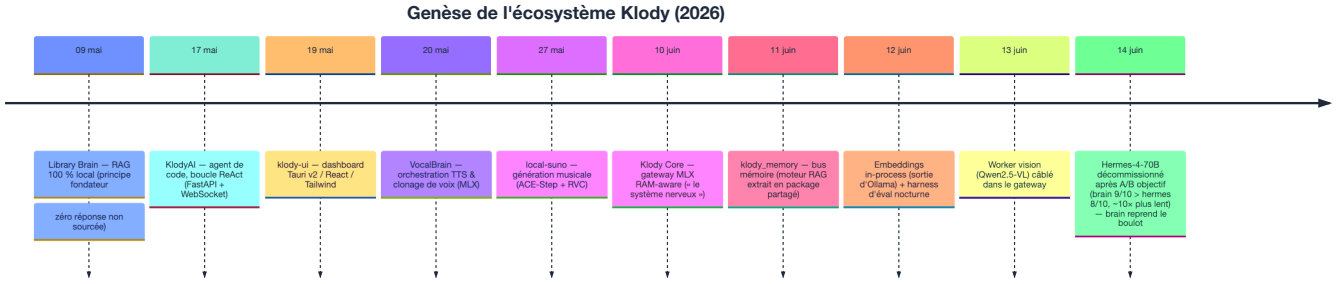
Compétence	Preuve concrète dans le code
Architecture systèmes	Identification d'un manque transverse (« 5 organes, aucun système nerveux ») → construction d'un *control plane*, pas d'une 6e feature.
Ingénierie de la concurrence	Scheduler asynchrone, verrou + double-vérification, suivi des requêtes en vol, nettoyage garanti par `BackgroundTask` (zéro fuite d'état).
Posture sécurité	Bind loopback imposé, borne anti-OOM sur les corps, gardes anti-DoS (`nan` / `inf`), télémétrie coupée — *threat-model* d'un service local.
Refactoring discipliné	Extraction d'un package de 11 modules par injection de dépendances + *shims* de rétrocompatibilité, validée par ~1 100 tests + une *gate* qualité après chaque sous-lot (pattern *strangler fig*).
Qualité ML / LLMOps	Distinction explicite « le code marche » (tests unitaires) vs « les modèles répondent bien » (harness d'éval nocturne, baselines épinglées, détection de régression).
Discipline de mesure	RAM mesurée au `vmmap` (pas au `ps rss` — connaissance fine de la mémoire unifiée Metal) ; bascule d'embeddings validée par `cos = 1.0000` **avant** d'éviter de ré-indexer 844 000 vecteurs.
Maturité opérationnelle	Tout est service `launchd` (RunAtLoad + KeepAlive), venvs dédiés pour casser les dépendances circulaires, endpoints de santé, logs structurés, éval cron.

Échelle : ~**381 commits** sur 6 dépôts, ~**2 350 tests** au vert, ~**4 000 lignes** pour le seul Core (gateway + package mémoire).

---

1. Genèse — la chronologie

L'écosystème n'a pas été conçu d'un bloc : il a **émergé**. Cinq applications autonomes d'abord, puis la prise de conscience qu'elles partageaient un manque commun, et enfin l'infrastructure qui les fédère.



Le déclic architectural (10 juin)

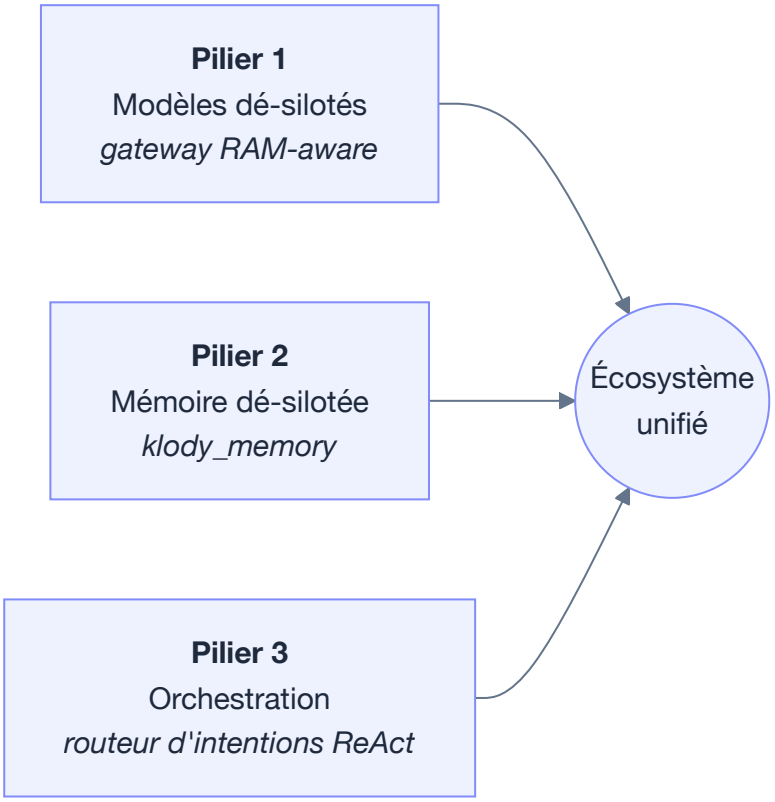
Le constat fondateur, formulé puis validé : *les cinq projets sont d'excellents organes, mais sans système nerveux.* Chacun chargeait ses propres modèles, gérait sa propre mémoire, exposait son propre service sur son propre port. Conséquence mesurée : **~208 Go de poids redondants sur disque** et **N inférences dupliquées** en RAM.

La décision déterminante — et la marque d'une réflexion d'architecte — fut de **ne pas réécrire les organes mais de les *fronter***. Klody Core est un *control plane* personnel, assumé comme tel (« hardcodé pour ~5

services, pas un framework »). L'effet est multiplicatif : *chaque heure investie dans le Core valorise tout l'existant.*

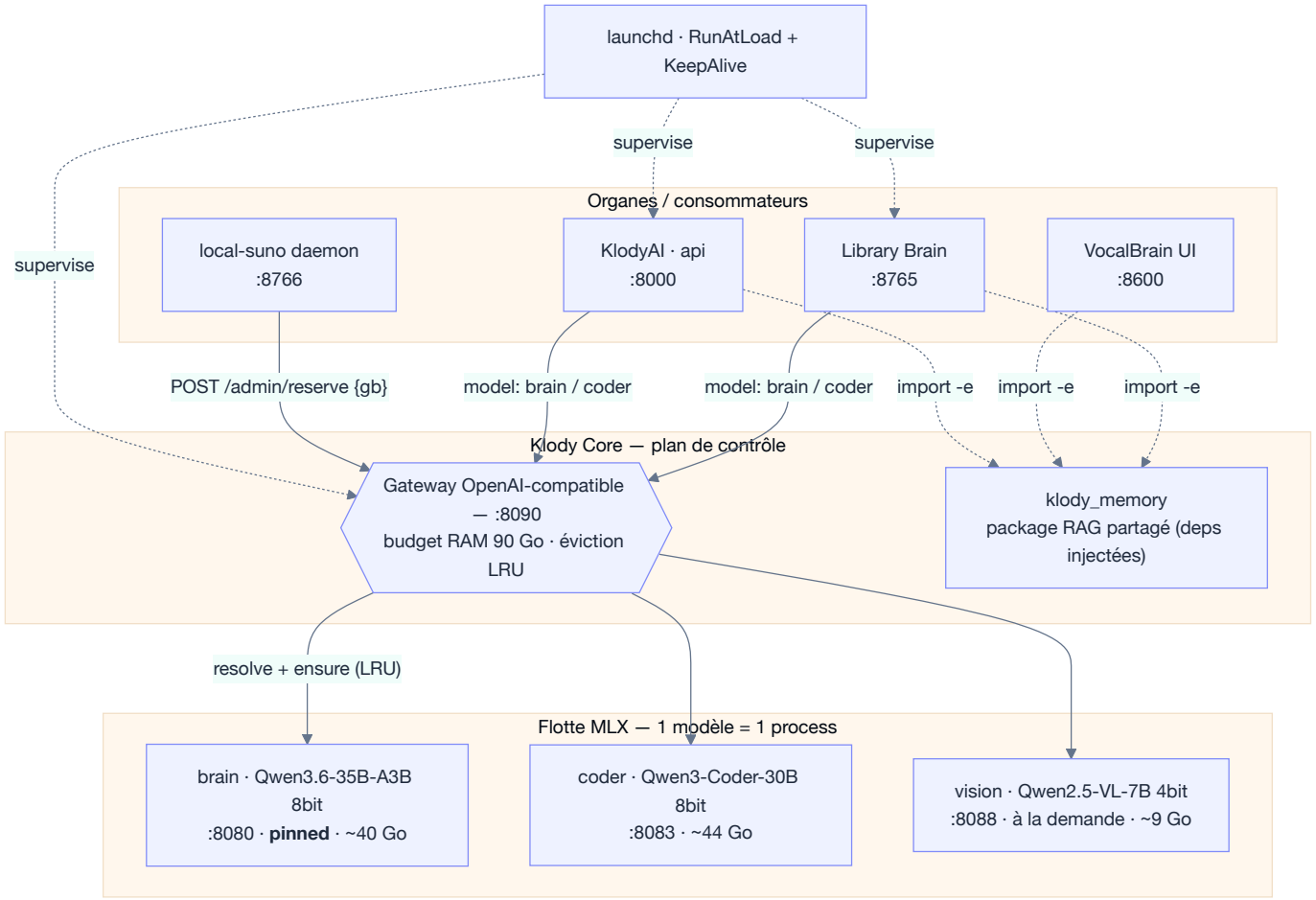
2. Vue d'ensemble — la thèse « système nerveux »

Klody Core repose sur **trois piliers**, attaqués dans l'ordre et tous livrés :



1. **Le gateway** — un endpoint d'inférence unique, partagé, conscient de la RAM.
2. **Le bus mémoire** — le moteur RAG promu en package réutilisable.
3. **Le routeur d'intentions** — l'agent ReAct qui pilote les organes (parler, chercher dans les livres, composer, se souvenir).

Topologie d'exécution



Deux flux distincts cohabitent :

- **Flux d'inférence** (trait plein) : les organes parlent OpenAI standard au gateway, qui résout l'alias, garantit la résidence du modèle et relaie (streaming compris).
- **Flux mémoire** (pointillé) : trois applications **importent** le même package `klody_memory` (`pip install -e`) — pas de pont HTTP, identité partagée du code.
- **Flux de réservation** : un job audio (ACE-Step/RVC) demande au gateway de **libérer de la RAM** avant de s'exécuter.

---

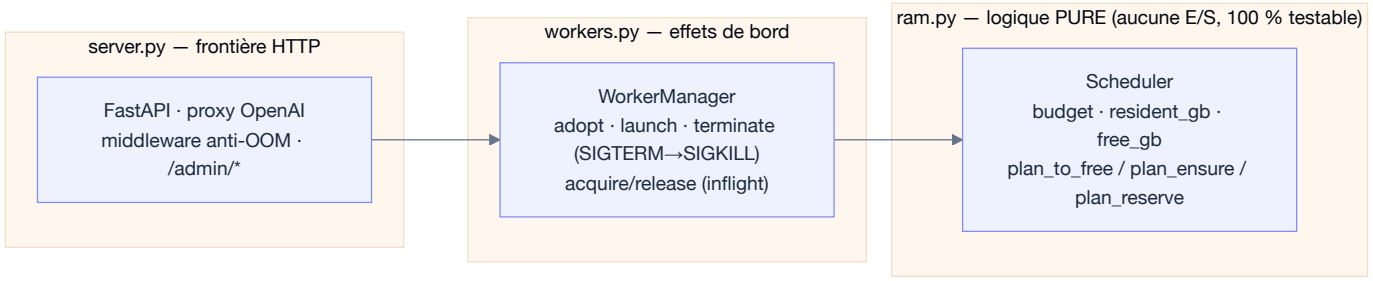
3. Pilier 1 — le gateway MLX *RAM-aware*

Le problème, précisément

`mlx_lm.server` (la stack d'inférence Apple Silicon) sert **un seul modèle figé par process** : pas de *swap* à chaud natif. Avant le Core, chaque application lançait/épinglait son propre serveur → le `brain` (~40 Go) et le `coder` (~44 Go) coexistaient en double, plus Ollama. Sur une machine à 128 Go, c'est le mur.

La solution : séparer la décision de l'effet

C'est le choix de conception le plus important du projet, et il est *textbook* : **la logique pure de planification est isolée de toute entrée/sortie.**



- `ram.py` ne contient **aucun** subprocess, aucune socket : juste des décisions (« quels workers évincer pour faire tenir `X` Go ? »). Il se teste hors-ligne, intégralement.
- `workers.py` applique ces plans au réel (lance/tue les process, suit les requêtes en vol).
- `server.py` est la frontière HTTP (proxy OpenAI + admin).

Cette séparation **hexagonale** est ce qui distingue un service jetable d'un service maintenable.

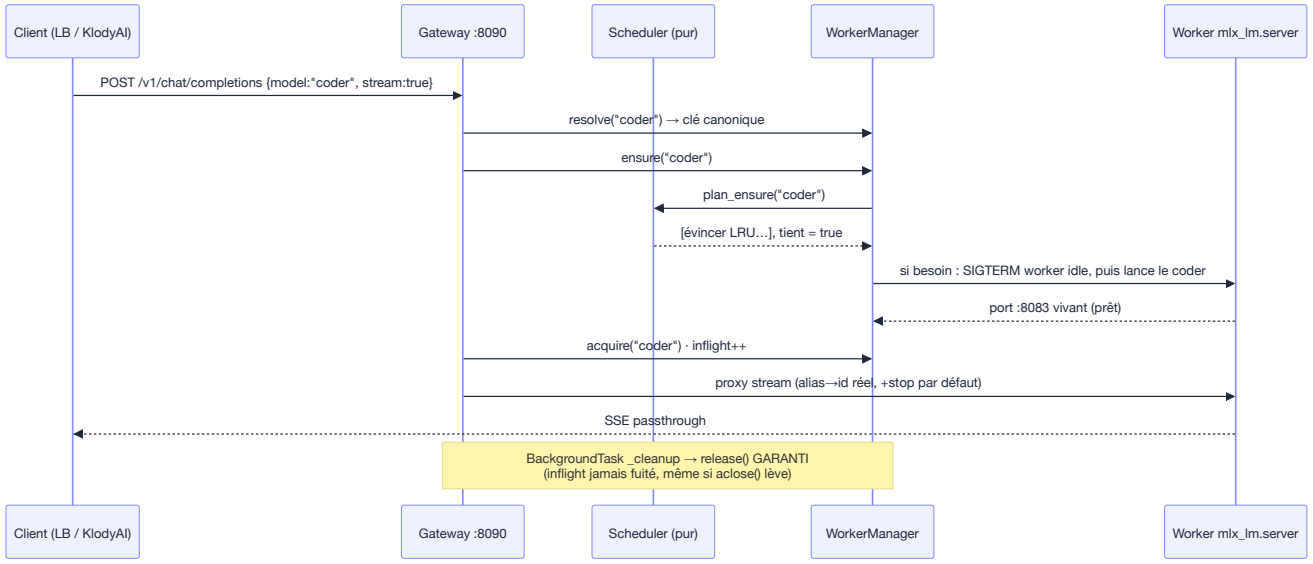
Mécanismes clés

Budget & éviction LRU. Budget de **90 Go** sur 128 (marge OS + jobs audio). Le `brain` est `pinned` (partagé par Library Brain *et* KlodyAI → jamais évincé) ; les autres workers sont évincés **du moins récemment utilisé**, jamais un worker `pinned` ni un worker servant une requête (`inflight > 0`).

Adoption. Au démarrage, le gateway **détecte** les workers déjà vivants sur leurs ports et les *adopte* au lieu de les relancer — un redémarrage du gateway ne tue pas l'inférence en cours.

Réservation pour l'audio. `POST /admin/reserve {"gb": 24}` libère la RAM *avant* qu'ACE-Step/RVC ne démarre. Le bilan renvoyé reflète la **RAM réellement libre** (`satisfied`), pas le plan — l'appelant doit vérifier avant de lancer son job.

Séquence d'une requête (streaming)



Détails qui trahissent le niveau

Ces choix ne s'inventent pas sans cicatrices — ils sont la signature d'un ingénieur expérimenté :

- **Pas de fuite d'`inflight` en streaming.** Le `release()` passe par un `BackgroundTask` Starlette, exécuté dans *tous* les chemins (fin normale, déconnexion client, annulation). Sans ça, si `aclose()` levait sur un transport cassé, le `coder` restait `inflight > 0` **à vie** → définitivement non-évinçable, RAM épinglée pour toujours.
- **Comptabilité RAM honnête.** Si un worker survit à `SIGTERM` *puis* `SIGKILL`, `_terminate` **restaure l'état réel et lève** au lieu de prétendre avoir libéré la RAM — mentir ici provoquerait un OOM au job audio suivant.
- **Course fermée tôt.** `running = False` est posé **avant** le premier `await` du `terminate`, sous le lock, pour fermer le chemin rapide de `ensure()` pendant l'extinction.
- **Relais brut en non-streaming.** On ne reparse pas la réponse du worker (`r.json()`) : un worker en erreur peut renvoyer du HTML/texte/vide → un re-parse lèverait et masquerait le vrai statut derrière un 500 opaque.
- **`vmmap`, pas `ps rss`.** Les poids MLX vivent dans des buffers Metal (mémoire unifiée) → `ps rss` affiche ~400 Mo pour un modèle de 40 Go. L'empreinte est mesurée au `vmmap -summary` (Physical footprint).

Extensibilité prouvée

Le registre (`config.py`) est purement déclaratif. Deux extensions récentes l'ont validé sans toucher au cœur :

- **Hermes-4-70B** (`:8091`, *depuis décommissionné*) : worker à la demande qui a **prouvé l'extensibilité du registre** avant d'être retiré. Piège non trivial résolu au passage — l'`eos_token_id` du dépôt (`128001`) ne couvrait pas le `<|eot_id|>` (`128009`) de fin de tour → génération *runaway* ; correctif robuste conservé dans le gateway : injection de `stop: ["<|eot_id|>"]` par défaut (survit à un re-download du cache HF). **Retiré le 14 juin** après A/B objectif (`brain` 9/10 vs `hermes` 8/10 pour ~10× plus lent) : aucun gain, `brain` reprend le boulot. *Savoir retirer un composant qui ne prouve pas sa valeur est aussi une décision d'architecte.*
- **Worker vision** (`:8088`, Qwen2.5-VL) : démarré via `mlx_vlm.server` (et non `mlx_lm.server`) — le lanceur adapte ses arguments selon le module et tolère un téléchargement multi-Go au premier lancement.

---

4. Pilier 2 — le bus mémoire `klody_memory`

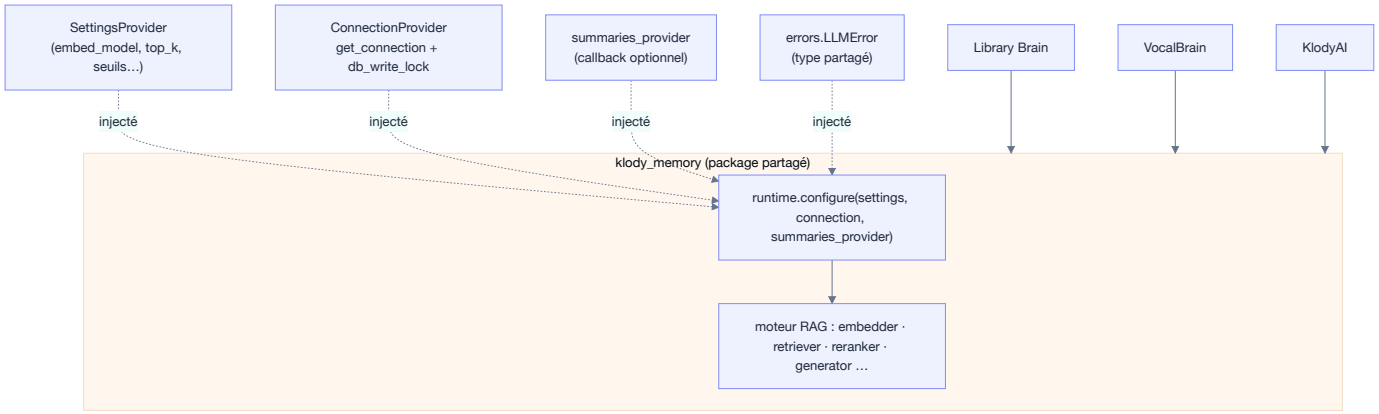
L'idée

Le moteur RAG le plus abouti vivait dans Library Brain. Plutôt que de le copier dans chaque app (dérive garantie) ou de l'exposer en HTTP (couplage réseau), je l'ai **extrait en package réutilisable à dépendances injectées** — la source de vérité reste unique, les consommateurs l'importent.

Méthode : *strangler fig* + injection de dépendances

L'extraction (11 modules : `models`, `embedder`, `retriever`, `reranker`, `generator`, `prompt_builder`, `scorer`, `translator`, `llm_provider`, `repo_retriever`, `web_search`) a suivi une **frontière définie *avant* de déplacer une ligne**, puis sous-lot par sous-lot, avec la suite de **~1 100 tests** et une **gate qualité RAG** rejouées après chaque étape.

Le *seam* est constitué de **4 dépendances injectées** via des `Protocol` Python :



Décision-clé : le package **ne possède pas la base de données** — il reçoit `get_connection` + `db_write_lock`. Cela règle d'office le problème du « second writer » SQLite : chaque consommateur garde sa propre base, son propre verrou, ses propres réglages.

Rétrocompatibilité : les anciens chemins `rag/*.py` de Library Brain deviennent des **shims** qui ré-exportent depuis `klody_memory` — zéro rupture, identité du code préservée.

Généricité prouvée par 3 consommateurs

Consommateur	Base dédiée	Domaine	Import Library Brain ?
Library Brain	`library_brain.db` (844 k chunks)	livres	— (source)
VocalBrain	`~/.vocalbrain/memory.db`	audio (« morceau triste au piano » → la bonne ballade)	**zéro**
KlodyAI	`logs/semantic_memory.db` (1 533 souvenirs)	sessions d'agent	**zéro**

Le fait que VocalBrain rappelle sémantiquement *le bon morceau* avec **zéro import** de Library Brain est la **preuve empirique** que l'abstraction est correcte.

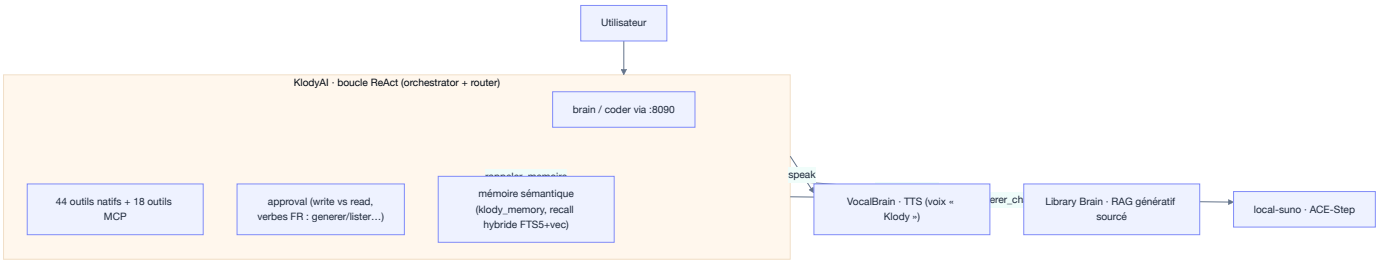
Décision de migration validée par la mesure

Les embeddings passaient par Ollama (daemon séparé). Avant de migrer vers `sentence-transformers` in-process, j'ai **mesuré** la compatibilité sur corpus réel (`measure_embed_compat.py`) : `cos(emb_ollama, emb_st) = 1.0000` partout → **zéro ré-encodage des 844 000 vecteurs existants**. Une migration risquée transformée en changement de configuration trivial, *parce que* mesurée d'abord.

---

5. Pilier 3 — le routeur d'intentions (agent ReAct)

Le gateway dé-silote les **modèles**, `klody_memory` dé-silote la **mémoire** — mais il fallait un **chef d'orchestre**. C'est **KlodyAI**, l'agent de code, promu routeur : sa boucle **ReAct** (Reason + Act) sélectionne et appelle des outils, dont les « bras moteurs » qui actionnent les autres organes.



Points remarquables :

- **Découverte de réutilisation** : à la mise en œuvre des « bras », 2 des 3 existaient déjà (le pont RAG et le pont musique via MCP). Le travail réel s'est concentré sur l'outil manquant — `speak` — et sur la **sécurité d'approbation** (les verbes français `generer/entrainer/supprimer…` étaient classés *write* et passaient sans validation).
- **Pièges TTS résolus empiriquement** : un texte multi-phrases d'un bloc → l'EOS ne venait jamais (163 s de WAV saturé) ; correctif = segmentation par `\n` (motif de split du modèle) + noms de langue complets (`french` , pas `fr`).
- **Mémoire sémantique** in-process (bge-m3, `sentence-transformers`), miroir automatique des faits/sessions, **1 533 souvenirs** *backfillés*, outil `rappeler_memoire` en langage naturel.

---

6. Les organes

Application	Rôle	Stack	État
Library Brain	RAG 100 % local — réponses *uniquement* depuis les livres (principe fondateur)	FastAPI · SQLite FTS5 + sqlite-vec · bge-m3 · ~5 800 livres / 844 k chunks	En prod (gate qualité au vert)
KlodyAI (`klody-code-ai` + `klody-ui`)	Agent de code ReAct, dashboard desktop	FastAPI · WebSocket · Tauri v2 · React · Tailwind v4	En prod (1 132 tests)
VocalBrain	Orchestration TTS & clonage de voix	Python · Pydantic v2 · mlx-audio · SQLite · routeur de modèles 4-D	En prod (38 tests)
local-suno	Génération musicale + conversion de voix	ACE-Step · RVC · daemon	En prod (37 tests)
RetroEcho	App macOS IA locale (pur-Swift)	Swift · MLX (LoRA)	Phases 1–8
GrainForge	Groovebox générative embarquée	Teensy 4.1 + Pi 4 + M5 Max	Phase 0 (matériel)

> Contrainte d'architecture assumée et tenue : **RetroEcho reste pur-Swift** — il consomme le gateway de façon *optionnelle*, jamais via un pont Python au runtime. Savoir où *ne pas* étendre une abstraction est aussi un signe de maturité.

---

7. Exploitation (Ops)

Carte des services `launchd`

Tout tourne en service supervisé (RunAtLoad + KeepAlive : démarre au login, se relance au crash).

Service `launchd`	Port	Rôle
`com.klody.core-gateway`	8090	gateway MLX
`com.klody.api`	8000	backend KlodyAI
`com.librarybrain.server`	8765	web Library Brain
`com.klody.vocalbrain-ui`	8600	UI VocalBrain
`com.klody.localsuno-daemon`	8766	daemon génération musicale
`com.klody.klody-mcp` / `web-mcp` / `gmail-mcp` / `vocalbrain-mcp`	8087 …	serveurs MCP (outils de l'agent)
`com.klody.nightly-eval`	—	éval qualité, **03 h 30** chaque nuit

Détails d'ingénieur : venv **dédié** `~/klody-core-env` pour **rompre la dépendance circulaire** (le gateway ne dépend plus du venv de Library Brain) ; le `PATH` de `launchd` doit inclure `/usr/sbin` (sinon `lsof` indisponible → détection de PID dégradée) — piège diagnostiqué et corrigé.

Le harness d'éval nocturne — la pièce de maturité

> Les tests unitaires prouvent que **le code** marche. Ils ne prouvent **pas** que **les modèles** répondent toujours bien. Cette distinction, rare même en équipe pro, est explicitement adressée.

`nightly_eval.py` (cron `launchd` 03 h 30) en trois sections indépendantes :

1. **Santé du gateway** — `/admin/status` + latence d'une complétion `brain`.
2. **Gate RAG complète** de Library Brain — `--check --generate` (retrieval **et** génération vs baseline épinglée) : keyword coverage, taux de refus, **zéro hallucination**.
3. **pass@1 code** — prompts *golden* → `coder` → le code généré est **exécuté sous pytest** (régression si < 2/3).

Rapport Markdown daté, **notification macOS uniquement en cas de régression** (le silence = tout va bien), code retour 1 si rouge.

Anecdote révélatrice (faux positif du 13 juin). Le premier run automatique est passé au rouge (gate RAG effondrée). Diagnostic : *pas une régression* — la nuit, `brain` (40) + `coder` (44) saturent les 90 Go ; à froid, charger les embeddings faisait **swapper** (brain ~13× plus lent). À chaud le matin : 100 % vert, même config. **Correctif** : décharger le `coder` (`/admin/unload`) *avant* la gate RAG nocturne — il ne sert qu'au pass@1. Savoir distinguer une régression d'un artefact d'environnement, c'est de l'expérience.

8. Méthodes & pratiques d'ingénierie

Catalogue des pratiques effectivement appliquées, observables dans le code et l'historique Git :

- **Séparation logique pure / effets de bord** (architecture hexagonale) → testabilité maximale.
- **Injection de dépendances par** `Protocol` → package découplé, 3 consommateurs hétérogènes.
- **Strangler fig** pour l'extraction de package → shims de rétrocompatibilité, zéro big-bang.
- **Test-as-you-go** : suite complète + gate qualité **après chaque sous-lot**, jamais en fin de course.
- **Revue adversariale pré-commit** (workflow multi-agents) avant le premier commit du gateway : **10 problèmes confirmés sur 22 candidats** (1 critique, 1 majeur, 4 moyens, 4 mineurs), tous tracés et corrigés (fuite d'`inflight`, comptabilité RAM mensongère, course de `running`, 500 opaques…).
- **Décisions pilotées par la mesure** : `vmmap` pour la RAM, `cos = 1.0` avant la bascule embeddings, tok/s mesurés avant d'adopter un modèle.
- **Comptabilité honnête / *fail loud*** : ne jamais prétendre un état non vérifié (RAM libérée, requête terminée).
- **Sécurité par défaut** : loopback imposé (refus de bind non-local sans opt-in explicite), borne anti-OOM, gardes `nan` / `inf` anti-DoS, télémétrie HF/MLX coupée.
- **Idempotence** : scripts de backfill / bootstrap relançables sans dégât.
- **Documentation du *pourquoi*** : READMEs et commentaires expliquent les arbitrages et les cas limites, pas le *quoi* trivial. Les notes de projet tiennent lieu d'**ADR** vivants.
- **Hygiène Git** : Conventional Commits, branche protégée sur `main` (PR + squash signé), backups GitHub privés.

---

9. Décisions d'architecture (ADR — extraits)

> Reformulées au format ADR à partir de l'historique réel.

ADR-1 — Fronter, ne pas réécrire. *Contexte* : 5 organes, ~208 Go de poids redondants. *Décision* : control plane qui *adopte* les services existants. *Conséquence* : valorisation multiplicative de l'existant ; le Core reste petit et focalisé.

ADR-2 — Logique de scheduling pure, isolée des E/S. *Décision* : `Scheduler` sans subprocess ni socket ; `WorkerManager` applique les plans. *Conséquence* : cœur testable hors-ligne (`test_ram.py`), cycle de vie testé séparément en hermétique (mocks).

ADR-3 — Le package mémoire ne possède pas la base. *Décision* : injecter `get_connection` + `db_write_lock`. *Conséquence* : pas de conflit « second writer », chaque consommateur garde sa base/ses réglages.

ADR-4 — Embeddings in-process plutôt que daemon. *Décision* : `sentence-transformers` chargé 1×/process (validé `cos = 1.0`). *Conséquence* : un daemon en moins sur le chemin critique ; encodage à chaud ~10 ms ; zéro ré-indexation.

ADR-5 — Distinguer « le code marche » de « le modèle répond bien ». *Décision* : harness d'éval nocturne avec baselines épinglées, séparé des tests unitaires. *Conséquence* : détection des régressions *qualité* (hallucination, couverture) invisibles aux tests classiques.

---

10. Évaluation du niveau de développement

> Section d'auto-évaluation **étayée par des preuves**, destinée au lecteur (recruteur, CTO, pair).

L'ensemble du travail situe l'auteur à un niveau **ingénieur logiciel senior / staff**, avec une **spécialisation infrastructure d'IA locale (LLMOps Apple Silicon)**. Les signaux, mappés à des compétences reconnaissables :

Signal observé	Compétence démontrée	Niveau
Reconnaître un manque transverse et construire un *control plane* plutôt qu'une feature	Pensée systèmes, vision d'architecture	Staff
Scheduler pur séparé des effets (hexagonal)	Conception pour la testabilité	Senior+
Fuite d'`inflight`, course de `running`, nettoyage par `BackgroundTask`	Concurrence asynchrone correcte	Senior+
Loopback imposé, anti-OOM, gardes anti-DoS, télémétrie coupée	Threat-modeling, sécurité par défaut	Senior
Extraction de package par DI + Protocols + shims, validée par 1 100 tests	Refactoring à grande échelle sans rupture	Senior+
Harness d'éval nocturne, baselines épinglées, pass@1 exécuté	Qualité ML / LLMOps	Rare (Senior/Staff ML)
`vmmap` vs `ps rss`, `cos = 1.0` avant migration	Discipline de mesure, profondeur système	Senior
Diagnostic du faux positif nocturne (swap, pas régression)	Débogage de systèmes en production	Senior+
`launchd`, venvs dédiés, endpoints santé, logs, cron	Maturité opérationnelle / SRE	Senior
Savoir où *ne pas* étendre (RetroEcho pur-Swift)	Jugement d'architecture	Senior+

Synthèse en une phrase pour un CV / une fiche de poste :

> *Ingénieur capable de concevoir et d'exploiter une infrastructure d'inférence LLM multi-modèles consciente des ressources, d'extraire des abstractions réutilisables sans casser l'existant, et d'instituer une assurance qualité spécifique aux systèmes IA — le tout en environnement 100 % local, contraint en mémoire, et mis en production.*

Domaines d'expertise étayés : architecture logicielle · systèmes distribués (à l'échelle d'une machine) · LLMOps / inférence MLX · RAG (retrieval, reranking, embeddings cross-lingues) · agents (ReAct, outils, MCP) · concurrence asyncio · sécurité applicative · DevOps macOS (`launchd`) · DSP/audio (TTS, voice cloning) · desktop (Tauri/React) · C++/JUCE (projets audio adjacents).

11. Métriques du projet

Dépôt	Premier commit	Commits	Tests	Rôle
`library-brain`	09 mai	150	~1 113	RAG local
`klody-code-ai`	17 mai	137	1 132	agent ReAct
`klody-ui`	19 mai	31	—	dashboard Tauri
`vocalbrain`	20 mai	23	38	TTS / voix
`local-suno`	27 mai	22	37	musique
`klody-core`	10 juin	18	28	plan de contrôle
Total	—	**~381**	**~2 348**	—

Autres chiffres : Klody Core ≈ **4 000 lignes** (gateway ~750 + `klody_memory` ~3 300) · flotte de **3 modèles** (35B/30B/7B-VL) · **844 738 chunks** indexés · **~208 Go** de redondance éliminée par mutualisation · `brain` 36,7 Go mesuré / `coder` 42,7 Go (pic 62,3).

12. Axes de progression / limites assumées

> Les nommer est volontaire : un portfolio crédible montre aussi la lucidité sur ses propres limites.

- **Mono-machine, pas d'échelle horizontale** — par conception (control plane *personnel*), mais le gateway n'est pas multi-nœuds.
- **Gateway sans authentification** — mitigé par le bind loopback ; un usage réseau exigerait une couche d'auth (la base est déjà posée : refus de bind non-local sans opt-in).
- **`est_ram_gb` statiques** — estimations mesurées mais figées ; une mesure dynamique (`footprint` en continu) raffinerait l'éviction.
- **Tension RAM tranchée par l'A/B** — la cohabitation `brain` + `coder` + `hermes` (40+44+44) dépassait 90 Go ; l'A/B objectif du harness a réglé le dilemme en **retirant** `hermes` (gain qualité nul pour un coût ~10×). La flotte `brain` + `coder` + `vision` tient désormais sous budget — la tension est résolue, pas seulement contournée.
- **Dettes mineures tracées** : `pyproject` du package à `dependencies=[]` (assumé, phase ultérieure) ; quelques mémoïsations globales non *thread-safe* (notées à l'audit) ; index FTS `chunks_fts` à reconstruire (`rebuild`).
- **Couplage métier du moteur RAG** : le retriever impose un modèle « source → chunks » (une entité = une ligne `books`) — généricité réelle mais bornée.

13. Stack technique (inventaire)

Langages : Python 3.11 · TypeScript/React · Swift · C++17/JUCE (audio). **Inférence** : MLX / `mlx_lm` 0.31.3 · `mlx_vlm` · `mlx-audio` · `sentence-transformers` · Ollama (legacy embeddings/healthcheck). **Modèles** :

Qwen3.6-35B-A3B (brain) · Qwen3-Coder-30B-A3B (coder) · Qwen2.5-VL-7B (vision) · bge-m3 (embeddings) · Qwen3-TTS / Irodori / MOSS (TTS) · whisper-large-v3-turbo (ASR, en cache). **Web / services** : FastAPI · Uvicorn · Starlette · httpx · WebSocket · Tauri v2 · Tailwind v4. **Données** : SQLite + **FTS5** + **sqlite-vec** · Pydantic v2. **Audio** : ACE-Step · RVC · librosa · soundfile · pyloudnorm. **Outils/agent** : protocole **MCP** · boucle ReAct maison. **Ops** : `launchd` (RunAtLoad/KeepAlive) · venvs dédiés · pytest · Git (Conventional Commits, branches protégées).

---

14. Glossaire

- **Gateway** — endpoint d'inférence unique (`:8090`) qui possède les workers et route par modèle.
- **Worker** — un process `mlx_lm.server` / `mlx_vlm.server` servant **un** modèle.
- `pinned` — modèle résident en permanence, jamais évincé (le `brain`).
- **Éviction LRU** — libérer la RAM en retirant le worker le moins récemment utilisé.
- `inflight` — nombre de requêtes en cours sur un worker (jamais évincé si > 0).
- **Adoption** — détecter et reprendre un worker déjà lancé au démarrage du gateway.
- `reserve` — libérer de la RAM à l'avance pour un job audio externe.
- `klody_memory` — package RAG réutilisable extrait de Library Brain.
- **ReAct** — boucle d'agent *Reason + Act* (raisonner puis appeler un outil).
- **MCP** — *Model Context Protocol*, standard d'exposition d'outils aux agents.
- **Gate / éval** — vérification automatique que les **modèles** répondent toujours bien (≠ tests unitaires du code).

Document de portfolio — synthèse de l'architecture Klody, de sa genèse (9 mai 2026) à son état du 14 juin 2026.