SEMANTIC COMPILER PLATFORM

Spécification Technique Complète

Architecture · Fonctionnalités · Implémentation · Déploiement

Infrastructure d'automatisation déterministe universelle

Version 1.0 — 2025

1. Vision et Positionnement Stratégique

1.1 Le Problème Fondamental

Les systèmes d'automatisation actuels obligent à choisir entre deux approches incompatibles. D'un côté, les agents LLM (LangGraph, n8n avec IA, OpenClaw) sont flexibles et capables de raisonner sur des cas imprévus, mais leur comportement est probabiliste, non certifiable et potentiellement dangereux dans des contextes critiques. De l'autre, les automates industriels classiques (PLCs, RPA) sont déterministes et certifiables, mais rigides, coûteux à programmer et inaccessibles aux utilisateurs non-techniques.

Cette tension crée un vide béant dans le marché : des secteurs entiers — médical, industriel, agricole, infrastructure critique — ne peuvent adopter l'IA car ils ne peuvent se permettre le non-déterminisme, mais ne peuvent pas non plus supporter la rigidité et le coût des systèmes classiques.

La Proposition de Valeur Centrale

Le Semantic Compiler Platform résout cette tension en utilisant le LLM comme compilateur statique, pas comme décideur dynamique. L'intelligence est confinée à la phase de compilation, supervisée humainement, et formellement vérifiée. L'exécution est purement déterministe, rapide et certifiable.

1.2 La Distinction Fondamentale

Dimension	Systèmes agents (LangGraph, n8n)	Semantic Compiler Platform
LLM au runtime	Systématique à chaque étape	Zéro — confiné à la compilation
Déterminisme	Probabiliste, variable	Absolu par construction
Certifiabilité	Impossible	IEC 61508, MDR, FDA possible
Déploiement edge	Serveur Linux minimum	Microcontrôleur 64KB RAM
Validation humaine	Inexistante	DAG validé avant exécution
Hallucinations runtime	Possibles	Structurellement impossibles
Audit trail	Logs applicatifs	Cryptographiquement signé
Utilisateur cible	Développeurs Python/JS	Métiers non-techniques

1.3 La Position Unique sur le Marché

Le Semantic Compiler Platform crée une nouvelle catégorie qu'aucun concurrent ne peut adresser avec son architecture actuelle :

- n8n / Make / Zapier : automatisation SaaS cloud, pas de déploiement edge, pas de certifiabilité, LLM non déterministe
- LangGraph / LlamaIndex : frameworks pour développeurs, LLM décideur au runtime, comportement probabiliste
- PLCs industriels (Siemens, Rockwell) : déterministes mais programmation spécialisée, coût élevé, pas de langage naturel
- OpenClaw : assistant personnel agentique, non déterministe, pas certifiable, pas de déploiement embarqué

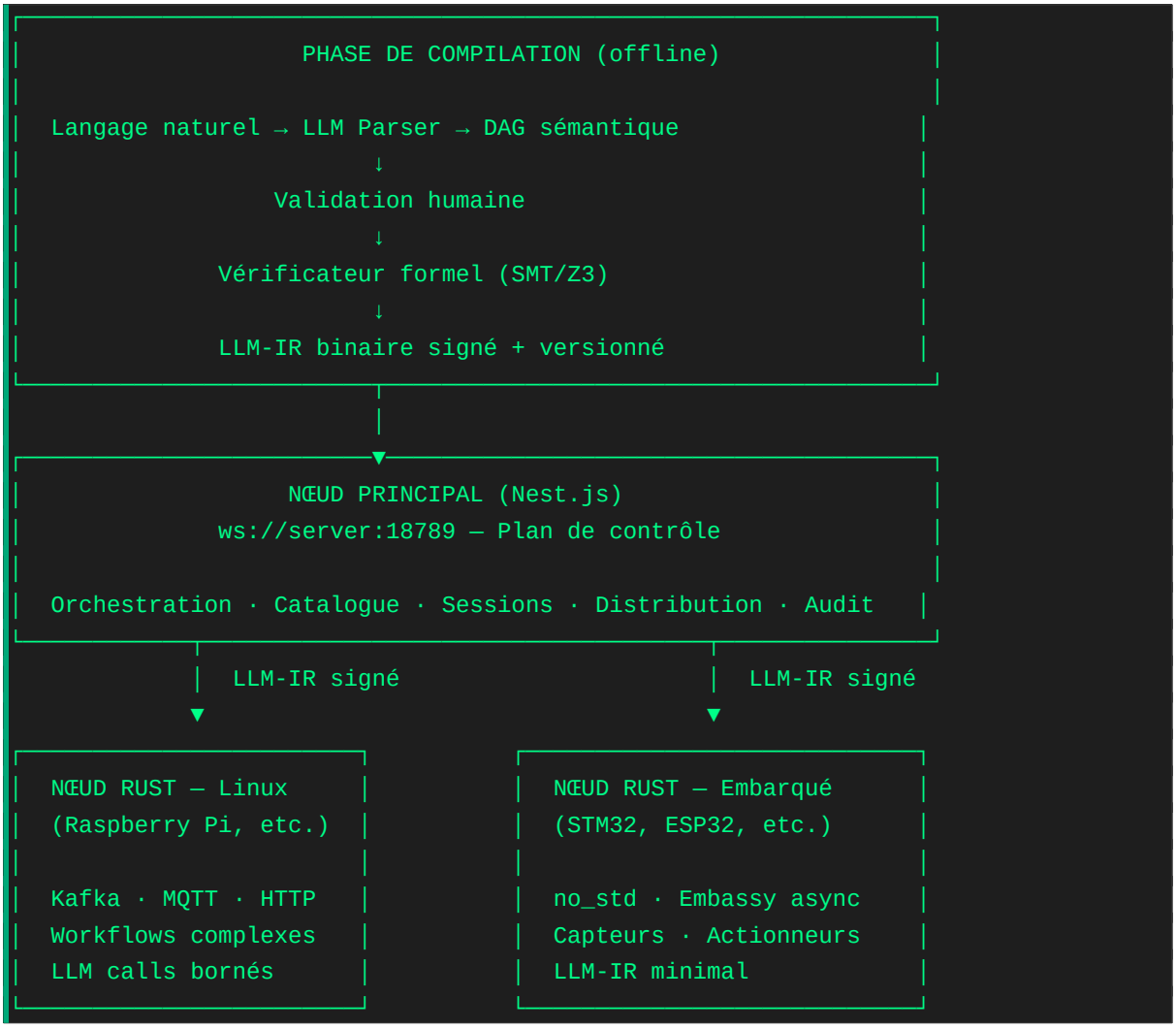
Positionnement Unique

"Le seul système qui permet à un opérateur non-technique de décrire en langage naturel un workflow critique, de le valider visuellement, et de l'exécuter de façon identique sur un microcontrôleur industriel comme sur un serveur cloud — avec la même garantie de déterminisme, de sécurité et d'auditabilité."

2. Architecture Globale du Système

2.1 Vue d'Ensemble

Le système est structuré autour de deux phases distinctes et d'un runtime distribué. Cette séparation est la garantie architecturale fondamentale du déterminisme.



2.2 Les Deux Cerveaux du Système

Le système implémente une architecture "neurosymbolique" avec une séparation formelle entre deux modes de traitement :

Mode	Description
Cerveau déterministe (SVM)	Pour les cas connus et routiniers. Exécution purement mécanique du LLM-IR compilé. Zéro LLM. Latence 10-50ms. Garantie absolue de comportement identique.
Cerveau raisonneur (LLM borné)	Pour les cas imprévus non couverts par les fallbacks compilés. LLM appelé avec contexte borné pré-construit, output validé formellement avant exécution. Max 3 tentatives puis escalade humaine.

2.3 Stack Technologique

Composant	Technologie	Justification
Nœud principal	NestJS (Node.js ≥22)	Écosystème riche, DI, WebSocket, BullMQ natif, développement rapide
Nœuds d'exécution	Rust (stable)	Pas de GC, sécurité mémoire, binaire autonome, embarqué possible
Format LLM-IR	Protocol Buffers (protobuf)	Binaire typé versionné, compact, cross-langage
Bus événements	Apache Kafka	Persistance, replay, exactly-once, découplage producteur/consommateur
Async embarqué	Embassy (Rust)	async/await sur microcontrôleur, pas de RTOS
Vérification formelle	Z3 SMT Solver (Rust binding)	Preuve formelle de terminaison et de contraintes
Transport LLM-IR	WebSocket + TLS	Reconnexion automatique, faible overhead
Audit log	Append-only + SHA-256	Inaltérable, certifiable réglementairement
Gestion secrets	HashiCorp Vault / injection runtime	Secrets jamais exposés au compilateur ou stockés en LLM-IR

3. Compilation Sémantique — Le Cœur du Système

3.1 Philosophie de la Compilation

La compilation sémantique est le processus qui transforme une intention exprimée en langage naturel en un programme binaire déterministe validé. Elle opère en plusieurs phases successives, chacune réduisant progressivement l'ambiguïté et augmentant le niveau de garantie formelle.

> **Principe Fondamental**
> Le LLM ne décide jamais pendant l'exécution. Il propose pendant la compilation. La machine décide si la proposition est formellement valide. L'humain valide ce qui sera fait. Ce triptyque — LLM propose, formel valide, humain approuve — est la garantie contre les hallucinations à impact réel.

3.2 Les Phases de Compilation

1. Phase 1 : Analyse Linguistique (LLM Parser)

Le LLM Parser reçoit la requête en langage naturel et produit une représentation structurée intermédiaire. À cette étape, la génération est contrainte par la grammaire du catalogue disponible — le LLM ne peut pas générer d'instructions référençant des connecteurs inexistants.

```
Entrée  : "Si la pression dépasse 8.5 bar, fermer la vanne V3 et alerter"
Sortie  : AST structuré {trigger: capteur_pression, condition: "> 8.5 bar",
          actions: [fermer_vanne_V3, alerter_operateur],
          type: event_driven, criticite: haute}
```

2. Phase 2 : Résolution du Catalogue

Le compilateur mappe les éléments de l'AST vers les entrées concrètes et typées du catalogue. Chaque connecteur, skill et service est résolu avec ses métadonnées complètes : préconditions, postconditions, contraintes de sécurité, latence estimée.

3. Phase 3 : Construction du DAG Sémantique

Le Directed Acyclic Graph est construit avec l'ensemble des chemins nominaux ET des branches de fallback. L'utilisateur est invité à sélectionner sa stratégie pour chaque cas imprévu potentiel identifié lors de la compilation.

4. Phase 4 : Validation Humaine

Le DAG est présenté à l'utilisateur sous forme visuelle et lisible. Il voit exactement ce que le système fera, dans quel ordre, avec quelles données, et quelle stratégie s'applique en cas d'anomalie. Il peut modifier, enrichir ou rejeter.

5. Phase 5 : Vérification Formelle

Une fois validé humainement, le DAG passe par le vérificateur formel (Z3 SMT Solver). Ce vérificateur garantit : terminaison du programme (pas de boucle infinie), respect des contraintes de ressources, cohérence des types entre les nœuds, satisfaction des préconditions/postconditions.

6. Phase 6 : Génération du LLM-IR et Signature

Le DAG validé et vérifié est compilé en LLM-IR binaire au format Protocol Buffers. Ce binaire est signé cryptographiquement (SHA-256 + clé privée de l'instance) et versionné. Le nœud Rust refuse d'exécuter tout LLM-IR non signé ou dont la signature ne correspond pas.

3.3 La Génération Contrainte

Pour garantir l'absence d'hallucinations dès la compilation, le LLM opère en mode de génération contrainte. Les tokens invalides — c'est-à-dire les références à des connecteurs ou actions non présents dans le catalogue — sont masqués pendant la génération. Le LLM ne peut physiquement pas générer un programme qui référence une ressource inexistante.

```
// La grammaire LLM-IR est définie par le catalogue
// Seuls les tokens valides peuvent être générés à chaque position
allowed_tokens_at_position = catalog.valid_next_tokens(current_context)
// Les tokens hors catalogue sont masqués avec logit_bias = -100
```

3.4 Le Contexte LLM Construit à la Compilation

Contrairement aux systèmes classiques qui construisent le contexte LLM au runtime (source principale d'hallucinations), le compilateur construit statiquement le contexte optimal pour chaque nœud LLM_CALL du DAG. Ce contexte est figé dans le LLM-IR.

```
struct CompiledLLMContext {
  system_prompt: String,          // figé à la compilation
  few_shot_examples: Vec<Example>,// sélectionnés à la compilation
  output_schema: JsonSchema,      // validation formelle de la sortie
  model: ModelIdentifier,         // modèle optimal choisi à la compilation
  temperature: f32,               // calibré selon le besoin du nœud
  max_tokens: u32,                // borné formellement
  dynamic_slots: Vec<ContextSlot>,// emplacements pour données runtime
}
```

Au runtime, la SVM injecte uniquement les données dynamiques dans les slots pré-définis. Le LLM reçoit un contexte parfaitement calibré, ce qui réduit drastiquement le risque d'hallucination.

3.5 Les Boucles de Raisonnement Contrôlées

Pour les problèmes nécessitant un raisonnement itératif (diagnostic, analyse complexe), le compilateur génère des boucles de raisonnement bornées. Contrairement aux boucles agent classiques qui peuvent diverger indéfiniment, ces boucles ont des conditions de sortie formelles compilées dans le LLM-IR.

```
struct LoopInstruction {
  max_iterations: u8,             // jamais > 5 par défaut
  timeout_ms: u32,               // timeout absolu
  convergence_predicate: Predicate, // condition d'arrêt formelle
  context_enrichment: EnrichmentStrategy, // comment enrichir à chaque tour
  fallback: FallbackInstruction,  // si convergence non atteinte
}
```

4. Le Catalogue — Couche de Confiance du Système

4.1 Rôle et Philosophie

Le catalogue est la pierre angulaire de la sécurité du système. C'est un registre structuré, signé et versionné de tout ce que le système peut faire. Le LLM ne peut composer que des éléments du catalogue — il ne peut pas inventer de nouvelles capacités. Cette contrainte est la garantie fondamentale contre les hallucinations à impact réel.

4.2 Structure d'une Entrée de Catalogue

```
interface CatalogEntry {
  // Identité
  id: string;              // UUID stable
  name: string;            // Nom lisible
  version: SemVer;         // Versioning strict
  category: EntryCategory; // trigger | action | transform | llm_call | service

  // Contrat formel
  input_schema: JsonSchema;   // Types d'entrée stricts
  output_schema: JsonSchema;  // Types de sortie stricts
  preconditions: Predicate[]; // Conditions requises avant exécution
  postconditions: Predicate[];// Garanties après exécution réussie

  // Métadonnées de performance
  estimated_latency_ms: Range; // Min/max latence
  resource_cost: ResourceCost; // CPU, RAM, réseau
  rate_limits: RateLimit[];    // Limites d'appels

  // Sécurité
  required_permissions: Permission[];
  safety_constraints: SafetyConstraint[];
  is_reversible: boolean;
  requires_human_confirmation: boolean;

  // Signature
  author: string;
  signature: CryptoSignature; // Signé par le créateur
}
```

4.3 Catégories du Catalogue

Catégorie	Exemples	Caractéristiques
Triggers	Kafka consumer, FS watcher, IoT MQTT, Webhook, Cron, DB poll, WebSocket	Sources d'événements. Déclenchent l'exécution du workflow compilé.
Actions de contrôle	Fermer vanne, Activer pompe, Redémarrer service, Envoyer commande	Agissent sur le monde physique. Contraintes de sécurité maximales, fenêtres temporelles, rate limits.
Actions de données	INSERT SQL, Publish Kafka, Write file, POST API, Send email/SMS	Modifications de données. Postconditions de vérification obligatoires.
Transformations	OCR, Parser PDF, Filtrage, Agrégation, Format conversion, Extraction	Traitements purs. Pas d'effets de bord. Parallélisables.
LLM Calls	Claude Opus, GPT-4o, Modèles spécialisés, Modèles locaux	LLMs comme services bornés. Contexte pré-construit. Output schématisé et validé.
Services métier	SAP, CRM Salesforce, EHR médical, ERP agricole, SCADA industriel	Connecteurs vers systèmes enterprise. Authentification gérée par Vault.
Lectures	Read sensor, Query DB, Fetch API, Read file, Consume event	Sources de données. Fréquence, cache, timeouts définis.

4.4 Catalogues Verticaux Spécialisés

Le catalogue est extensible par domaine. Chaque secteur vertical dispose de son catalogue spécialisé, développé et certifié par des experts métier :

- Catalogue Médical : connecteurs EHR/HIS, capteurs patient certifiés MDR, protocoles FHIR, actions médicales avec validation double
- Catalogue Industriel : protocoles Modbus, OPC-UA, MQTT industriel, capteurs certifiés IEC, actionneurs avec SIL
- Catalogue Agricole : capteurs sol/météo, systèmes d'irrigation, APIs météo, ERPs agricoles, rapports PAC
- Catalogue Finance : connecteurs bancaires PCI-DSS, APIs trading, systèmes de rapports réglementaires DORA
- Catalogue IoT Grand Public : appareils domestiques, APIs cloud (Google, Amazon), notifications mobiles

4.5 Système de Signature et Gouvernance

Chaque entrée de catalogue est signée cryptographiquement par son auteur. Le système de gouvernance permet de définir qui peut ajouter, modifier ou révoquer des entrées de catalogue selon le contexte de déploiement.

Rôle	Permissions
Administrateur système	Ajouter/révoquer des entrées dans tous les catalogues
Développeur certifié	Ajouter des entrées dans les catalogues de son domaine
Opérateur métier	Utiliser les entrées du catalogue pour créer des workflows
Auditeur	Lecture seule de l'historique du catalogue et des signatures

5. Le LLM-IR — Format Intermédiaire Déterministe

5.1 Philosophie du Format

Le LLM-IR (Large Language Model Intermediate Representation) est le format binaire dans lequel les DAGs compilés sont stockés et distribués. Il s'inspire de la conception de LLVM IR — une représentation intermédiaire typée, versionnée et optimisable, qui découple la phase d'intelligence de la phase d'exécution.

Propriétés Fondamentales du LLM-IR

• Typé : chaque instruction a des types d'entrée et de sortie stricts, validés à la compilation • Versionné : incompatibilité de version = refus d'exécution par la SVM • Signé : signature cryptographique SHA-256, inaltérable • Déterministe : même LLM-IR = même comportement, toujours • Compact : Protocol Buffers, environ 2-10KB pour un workflow typique

5.2 Structure du LLM-IR

```
message LlmIR {
  string dag_id = 1;
  uint32 version = 2;
  bytes signature = 3;           // SHA-256 de l'ensemble
  string validated_by = 4;       // ID utilisateur validateur
  int64 compiled_at = 5;         // Timestamp compilation
  repeated Instruction instructions = 6;
  repeated Edge edges = 7;
  ExecutionPolicy policy = 8;
  repeated FallbackRule fallbacks = 9;
}
```

```
message Instruction {
  string id = 1;
  InstructionType type = 2;      // READ | TRANSFORM | CONTROL | LLM_CALL | ...
  string catalog_entry_id = 3;   // Référence catalogue versionnée
  bytes compiled_context = 4;    // Contexte pré-construit (pour LLM_CALL)
  repeated Constraint constraints = 5;
  repeated Predicate preconditions = 6;
  repeated Predicate postconditions = 7;
  optional LoopConfig loop_config = 8;
  optional TimeWindow allowed_window = 9;
  optional RateLimit rate_limit = 10;
  bool requires_audit_log = 11;
}
```

5.3 Versioning et Compatibilité

Le LLM-IR utilise un versioning sémantique strict. La SVM maintient une matrice de compatibilité :

Changement de version	Comportement SVM
Même version majeure	Exécution autorisée avec avertissement si version mineure différente
Version majeure différente	Refus d'exécution, demande de recompilation
Signature invalide	Refus absolu, alerte sécurité immédiate
Catalogue entry révoquée	Refus d'exécution, notification au nœud principal

5.4 Gestion du Stockage DAG

Le système maintient deux représentations synchronisées du DAG, avec une règle stricte de dérivation unidirectionnelle :

```
DAG JSON (source de vérité humaine)
  → Stocké en base de données versionnée (PostgreSQL)
  → Lisible et modifiable par les opérateurs
  → Contient le hash du LLM-IR correspondant
  ↓ (dérivation unidirectionnelle, jamais l'inverse)
LLM-IR binaire (source de vérité machine)
  → Stocké comme artifact signé
  → Jamais édité directement
  → Distribué aux nœuds Rust via canal sécurisé
```

6. La Semantic Virtual Machine (SVM) — Runtime de Haute Performance

6.1 Architecture de la SVM Rust

La SVM est le cœur exécutif du système, implémentée en Rust pour des garanties maximales de performance, sécurité mémoire et déterminisme. Elle interprète le LLM-IR et orchestre l'exécution des instructions sans jamais prendre de décision autonome.

```
pub struct SVM {
  ir_loader: IrLoader,           // Charge et valide le LLM-IR
  scheduler: Scheduler,          // Ordonnancement parallèle/séquentiel
  memory: ExecutionMemory,       // Gestion mémoire déterministe
  connector_registry: Registry,  // Connecteurs chargés depuis catalogue
  fallback_engine: FallbackEngine,// Gestion des cas imprévus
  audit_writer: AuditWriter,     // Écriture append-only cryptographique
  health_monitor: HealthMonitor, // Surveillance santé des connexions
  secret_vault: VaultClient,     // Injection secrets au runtime
}
```

6.2 L'Ordonnanceur (Scheduler)

L'ordonnanceur analyse les dépendances encodées dans le LLM-IR pour déterminer quelles instructions peuvent être exécutées en parallèle et lesquelles doivent être séquentielles. Cette analyse est statique — déterminée à la compilation, pas au runtime.

```
// Exemple d'exécution parallèle sûre
// Actions sans dépendance mutuelle → parallelisées automatiquement
PARALLEL {
  READ capteur_temperature     // aucune dépendance
  READ capteur_pression        // aucune dépendance
  FETCH meteo_api              // aucune dépendance
}
// Synchronisation sur les trois résultats
EVALUATE_CONDITIONS(temperature, pression, meteo)
// Actions dépendantes → séquentielles
IF condition_met:
  CONTROL fermer_vanne         // dépend de l'évaluation
  WRITE audit_log              // dépend de l'action de contrôle
  ALERT operateur              // dépend du log
```

6.3 Gestion de la Mémoire d'Exécution

La mémoire d'exécution est structurée en trois couches selon la durée de vie et les besoins de persistance :

Couche	Données	Persistance
État persisté	Compteurs de déclenchement, dernière exécution, historique conditions	Base de données append-only, survit aux redémarrages
Buffer pipeline	Output Action N → Input Action N+1, résultats intermédiaires	Mémoire uniquement, durée de vie = une exécution
Contexte d'exécution	User ID, permissions, project ID	Jamais persisté, réhydraté depuis source autoritaire à chaque exécution

6.4 Le Moteur de Fallback

Le moteur de fallback gère les situations où le chemin nominal ne peut pas être suivi. Toutes les stratégies de fallback sont définies à la compilation par l'utilisateur — il n'y a pas de décision autonome au runtime.

Stratégie	Comportement
FAIL_SAFE	Arrêt immédiat du workflow, alerte opérateur, attente intervention humaine. Pour systèmes critiques de sécurité.
DEGRADED_MODE	Exécution du chemin alternatif pré-compilé le plus sûr. Pour workflows métier non critiques.
RETRY_WITH_BACKOFF	N tentatives avec délai exponentiel avant escalade. Pour pannes transitoires (réseau, API).
LLM_REASONING	Activation du raisonneur LLM borné avec contexte pré-construit. Pour cas imprévus non couverts.
SUPERVISED_RECOMPILE	Déclenche une nouvelle phase de compilation avec notification à l'utilisateur. Pour évolution du contexte.

6.5 Gestion de la Concurrence sur Ressources Partagées

Quand plusieurs workflows compilés nécessitent la même ressource simultanément, la SVM applique la politique de priorité définie à la compilation — jamais une décision prise au runtime.

```
// Politique de priorité compilée dans le LLM-IR
struct PriorityPolicy {
```

```
  priority_level: u8,         // 0 = critique, 255 = basse priorité
  preemptible: bool,          // peut être interrompu par priorité plus haute
  max_wait_ms: u32,           // délai max avant fallback si ressource occupée
  fallback: FallbackInstruction,
}
```

7. Sources d'Événements et Déclencheurs

7.1 Architecture Event-Driven

Le système est fondamentalement event-driven. Contrairement aux outils comme n8n qui fonctionnent majoritairement en mode pull (polling périodique), la SVM maintient des listeners persistants pour chaque type de source d'événement, permettant une réaction instantanée.

> **Avantage Critique par Rapport au Polling**
>
> Un système basé sur le polling vérifie l'état toutes les N secondes — si un incident survient entre deux vérifications, le temps de réaction peut être fatal dans un contexte médical ou industriel. Les listeners event-driven de la SVM réagissent en millisecondes après l'événement.

7.2 Intégration Apache Kafka

Kafka est le bus principal pour les événements haute fréquence et les données IoT. L'intégration offre des garanties critiques pour les systèmes fiables :

- Exactly-once delivery : chaque événement est traité une et une seule fois, même en cas de crash du nœud
- Persistance des événements : les événements sont conservés, permettant le replay pour audit ou débogage
- Offset management : la SVM reprend exactement là où elle s'est arrêtée après un redémarrage
- Consumer groups : plusieurs nœuds peuvent consommer le même topic en parallèle pour la scalabilité

```
// Configuration Kafka compilée dans le LLM-IR
struct KafkaTrigger {
  bootstrap_servers: Vec<String>,
  topic: String,
  consumer_group: String,
  partition_assignment: PartitionStrategy,
  offset_reset: OffsetReset,          // earliest | latest | specific
  deserialization: DeserStrategy,      // JSON | Avro | Protobuf
  filter: Option<CompiledFilter>,      // pré-filtre avant trigger
  exactly_once: bool,
}
```

7.3 Sources d'Événements Supportées

Source	Protocole/Tech	Cas d'usage typiques
Apache Kafka	Kafka Consumer API	IoT haute fréquence, événements métier, logs système, flux de données
MQTT (IoT)	MQTT 3.1.1 / 5.0	Capteurs industriels, appareils connectés, domotique
Modbus	Modbus TCP/RTU	Automates industriels, PLCs, compteurs énergétiques
OPC-UA	OPC Unified Architecture	Équipements industriels haute fiabilité
File System	inotify (Linux), FSEvents (macOS)	Surveillance dossiers, traitement de fichiers entrants
HTTP Webhooks	HTTP/HTTPS POST	APIs externes, événements SaaS, intégrations cloud
WebSocket	WebSocket protocol	Flux de données temps réel, interfaces de contrôle
Cron / Schedule	Cron expression	Tâches périodiques, rapports planifiés, maintenances
Database CDC	Debezium / pg_logical	Changements en base de données (insert, update, delete)
Email (IMAP/Gmail)	IMAP / Gmail PubSub	Traitement automatique d'emails, factures, alertes
Signaux système	SIGTERM, SIGINT, custom	Événements système, déclencheurs de maintenance

8. Exécution Distribuée

8.1 Topologie Multi-Nœuds

Le système supporte une topologie distribuée flexible adaptée à chaque contexte de déploiement. Le nœud principal NestJS orchestre un ensemble de nœuds d'exécution Rust qui peuvent être déployés sur une variété de hardware.

```
Topologie type pour une usine intelligente :

Serveur central (Nest.js) — Cloud ou on-premise
          |
          ├── Nœud Rust #1 : Raspberry Pi 4 — Ligne production A
          |          └── Sous-nœuds STM32 : capteurs température/pression
          |
          ├── Nœud Rust #2 : Industrial PC — Salle de contrôle
          |          └── Connexion SCADA/OPC-UA
          |
          ├── Nœud Rust #3 : Raspberry Pi — Ligne production B
          |
          └── Nœud Rust #4 : ESP32 — Capteurs environnementaux
```

8.2 Protocole de Communication Nœud Principal ↔ Nœuds Rust

Le protocole de communication est conçu pour la résilience — les nœuds continuent de fonctionner même si la connexion au serveur central est temporairement interrompue.

Canal	Usage
WebSocket TLS (persistent)	Distribution des LLM-IR compilés, mises à jour de catalogue, commandes de gestion
HTTP REST (one-shot)	Requêtes de statut, health checks, demandes de logs
Kafka (événements)	Remontée des résultats d'exécution, métriques, audit events

8.3 Résilience et Mode Hors-Ligne

Chaque nœud Rust fonctionne de manière autonome une fois le LLM-IR reçu. En cas de perte de connexion avec le serveur central :

- Le nœud continue d'exécuter les workflows compilés stockés localement

- Les résultats d'exécution et les événements d'audit sont bufferisés en mémoire flash locale
- À la reconnexion, le nœud synchronise automatiquement son état avec le serveur central
- Les LLM_CALL nécessitant un accès externe sont mis en attente ou déclenche le fallback FAIL_SAFE selon la configuration compilée

8.4 Déploiement sur Objets Connectés

La SVM Rust est conçue pour être déployée sur une gamme très large de hardware, des serveurs aux microcontrôleurs :

Catégorie Hardware	Exemples	Capacités SVM
Serveurs / VMs	Linux x86_64, ARM64 servers	SVM complète, tous connecteurs, LLM_CALL local ou cloud
Linux embarqué	Raspberry Pi 4, NVIDIA Jetson, Siemens IOT2050	SVM complète, Kafka, MQTT, HTTP, LLM_CALL cloud
Microprocesseurs ARM	Raspberry Pi Zero 2W, BeagleBone	SVM standard, connecteurs légers, buffer local
Microcontrôleurs ARM Cortex-M	STM32, nRF52, SAMD51	SVM no_std, connecteurs I2C/SPI/UART, Embassy async
Microcontrôleurs WiFi/BLE	ESP32, ESP32-C3 (RISC-V)	SVM no_std, WiFi/MQTT, connecteurs IoT basiques

```
# Cross-compilation depuis machine de développement vers cible embarquée
# ARM Linux (Raspberry Pi)
rustup target add aarch64-unknown-linux-gnu
cargo build --target aarch64-unknown-linux-gnu --release

# ARM Cortex-M (STM32)
rustup target add thumbv7em-none-eabihf
cargo build --target thumbv7em-none-eabihf --release
# Binaire autonome, ~200KB, aucune dépendance runtime
```

9. Contrôle Physique — Actions sur le Monde Réel

9.1 Types d'Actions de Contrôle

Le système supporte trois catégories d'actions physiques, chacune avec des niveaux de garanties de sécurité adaptés à leur criticité :

Type	Exemples	Garanties de sécurité
Contrôle direct	Fermer vanne, Activer pompe, Ajuster température, Démarrer moteur	Fenêtres temporelles, rate limits, postconditions, fenêtre d'annulation
Commandes système	Redémarrer service, Déclencher procédure urgence, Changer état machine	Autorisations compilées, audit obligatoire, confirmation humaine si configuré
Actionneurs IoT	Relais, servomoteurs, vannes électriques, régulateurs	Contraintes physiques compilées (min/max), protection contre commandes invalides

9.2 Mécanismes de Sécurité pour le Contrôle Physique

Fenêtres temporelles d'autorisation

Chaque action de contrôle peut être limitée à des plages horaires précises, définies à la compilation et validées par l'utilisateur. La SVM vérifie systématiquement la fenêtre avant toute exécution.

```
struct TimeWindow {
  days: Vec<Weekday>,      // jours autorisés
  start: NaiveTime,        // heure de début
  end: NaiveTime,          // heure de fin
  timezone: Tz,            // timezone explicite
}
// Si hors fenêtre → fallback compilé, JAMAIS l'action
```

Fenêtre d'annulation pour actions irréversibles

```
async fn execute_with_cancellation_window(action, window_ms, cancel_rx) {
  notify_pending(&action).await;  // "INTENTION: fermer vanne dans 30s"
  tokio::select! {
    _ = sleep(Duration::from_millis(window_ms)) => execute_action(&action).await,
    _ = cancel_rx.recv() => ActionResult::Cancelled
  }
}
```

Vérification postcondition obligatoire

Après chaque action de contrôle physique, la SVM vérifie que l'état réel du système correspond à l'état attendu (postcondition). Si la vérification échoue, le fallback compilé se déclenche.

9.3 Exemple Complet : Workflow Industriel Hybride

Ce workflow traverse la frontière physique/numérique dans le même DAG compilé — une capacité unique au Semantic Compiler Platform :

```
DÉTECTER anomalie machine (capteur IoT Kafka)
     ↓
CRÉER ticket automatique (Jira REST API)
     ↓
NOTIFIER technicien (Slack + SMS)
     ↓
SI pas de réponse dans 15min:
  RÉDUIRE cadence machine (actionneur Modbus) ← monde physique
  [postcondition: cadence_mesurée < cadence_cible + 5%]
     ↓
COMMANDER pièce de rechange (ERP SAP)
     ↓
METTRE À JOUR planning production (Excel/SAP)
     ↓
INFORMER client concerné (email automatique)
```

10. Appels LLM Avancés — Intelligence Bornée

10.1 Le LLM comme Service Déterministe

Dans le Semantic Compiler Platform, les LLMs ne sont pas des décideurs — ils sont des processeurs sémantiques spécialisés, appelés comme des services avec des inputs typés et des outputs validés formellement. Cette distinction est fondamentale.

```
// LLM classique (agent) — dangereux
LLM reçoit contexte vague → décide quoi faire → exécute

// LLM dans SCP — contrôlé
SVM exécute instruction LLM_CALL {
  inject(dynamic_data, pre_built_context) → appelle LLM
  validate_output(output, output_schema)
  if invalid → retry(max=3) → fallback
  else → pass_typed_output_to_next_node()
}
```

10.2 Construction Statique du Contexte LLM

Le compilateur analyse chaque nœud LLM_CALL du DAG et construit le contexte optimal au moment de la compilation, pas au runtime. Ce contexte pré-construit est stocké dans le LLM-IR et injecté directement par la SVM lors de l'appel.

> **Pourquoi c'est révolutionnaire**
>
> Un LLM avec un contexte précis, borné et calibré hallucine infiniment moins qu'un LLM avec un contexte construit à la volée. Le compilateur peut sélectionner les few-shot examples les plus pertinents, calibrer la température selon le besoin (extraction = 0.0, raisonnement = 0.3), et définir exactement le format de sortie attendu. Tout ça est fait une fois, à froid, par un système qui a le temps de bien faire les choses.

10.3 Orchestration Multi-LLM

Le compilateur peut générer des DAGs qui orchestrent plusieurs LLMs différents en pipeline, chacun recevant exactement le contexte dont il a besoin et produisant un output typé consommé par le suivant :

```
// Pipeline multi-LLM compilé pour analyse médicale
READ rapport_imagerie (DICOM)
     ↓
LLM_CALL_1: Claude Opus
```

```
  context: [spécialité_radiologie, exemples_annotés, format_structuré]
  output_schema: {anomalies: [], confidence: float, zones: []}
     ↓ output validé
LLM_CALL_2: GPT-4o Vision (si confidence < 0.85)
  context: [analyse_précédente, zones_suspectes_zoomées, guidelines]
  output_schema: {validation: bool, confidence: float, notes: string}
     ↓ output validé
LLM_CALL_3: Modèle médical spécialisé
  context: [résultats_1_2, historique_patient, protocoles_établissement]
  output_schema: rapport_médical_structuré
     ↓
WRITE dossier_médical (EHR)
ALERT médecin_responsable (si anomalie critique)
```

10.4 Boucles de Raisonnement Itératif

Pour les problèmes nécessitant un raisonnement itératif (diagnostic, résolution de problèmes complexes), le compilateur génère des boucles bornées où chaque itération enrichit le contexte avec les résultats précédents.

```
BOUCLE diagnostic_panne (max_iterations=3, timeout=30s) {
  LLM_CALL: DiagnosticModel
    input: {donnees_capteurs, historique_pannes, iteration_precedente}
    output: {cause_probable, confidence, donnees_manquantes[]}

  SI confidence > 0.9 → SORTIR (succès)
  SI donnees_manquantes → COLLECTER donnees_manquantes → CONTINUER
  SI iteration >= max → FALLBACK: escalade_humaine
}
```

11. Versioning, Modificabilité et Gestion du Cycle de Vie

11.1 Système de Versioning

Le versioning est le mécanisme qui garantit la traçabilité complète de l'évolution des workflows. Il est conçu pour répondre à une exigence réglementaire fondamentale : pouvoir prouver, pour n'importe quelle exécution passée, avec quel programme exact elle a été réalisée, et qui l'a validé.

```
interface WorkflowVersion {
  task_id: UUID;          // Stable — identité du workflow
  version: number;        // Incrémental — identité du programme
  parent_version: number; // Traçabilité de l'évolution
  status: "active" | "archived" | "executing" | "deprecated";
  dag_json: object;       // Source de vérité humaine
  ir_checksum: string;    // Hash SHA-256 du LLM-IR correspondant
  validated_by: string;   // ID de l'utilisateur validateur
  validated_at: DateTime; // Timestamp de validation
  change_reason: string;  // Justification de la modification
}
```

11.2 Cycle de Vie d'une Modification

7. Demande de modification : l'utilisateur initie une modification du workflow actif
8. Vérification de l'état : la SVM vérifie si des exécutions sont en cours sur la version active
9. Gestion des exécutions actives : attendre la fin ou forcer l'arrêt propre avec audit
10. Nouvelle compilation : la version N+1 est compilée depuis le DAG modifié
11. Validation humaine : le nouvel DAG est présenté pour validation
12. Vérification formelle : Z3 vérifie les propriétés du nouveau programme
13. Signature et déploiement : le nouveau LLM-IR est signé et distribué aux nœuds
14. Archivage : la version N est archivée — immuable, jamais supprimée

11.3 Politique d'Archivage et d'Audit

Aucune version n'est jamais supprimée du système. Cette règle est fondamentale pour la conformité réglementaire. L'historique complet des exécutions est conservé avec référence à la version exacte utilisée.

Type de donnée	Politique de rétention
Versions des workflows (DAG JSON)	Conservation permanente, immuable après archivage
LLM-IR binaires signés	Conservation permanente, inaltérable
Logs d'exécution	Minimum 10 ans pour usage médical/industriel critique
Audit trail cryptographique	Conservation permanente, append-only
Résultats d'exécution	Configurable selon contexte (30 jours à permanent)

12. Logging, Audit et Observabilité

12.1 L'Audit Trail Cryptographique

L'audit trail est la couche de responsabilité légale et réglementaire du système. Chaque événement significatif est enregistré de façon append-only et signé cryptographiquement, garantissant qu'aucune entrée ne peut être modifiée ou supprimée après coup.

```
struct AuditEvent {
  event_id: UUID,
  timestamp: DateTime<Utc>,     // Précision nanoseconde
  node_id: String,              // Nœud qui a généré l'événement
  workflow_id: UUID,
  workflow_version: u32,        // Version EXACTE utilisée
  instruction_id: String,       // Instruction précise dans le LLM-IR
  event_type: AuditEventType,   // EXECUTION_START | ACTION_TAKEN | FALLBACK | ...
  input_hash: [u8; 32],         // Hash des données d'entrée
  output_hash: [u8; 32],        // Hash des données de sortie
  duration_ms: u64,
  result: ExecutionResult,
  previous_event_hash: [u8; 32],// Chaîne cryptographique (blockchain-like)
  signature: CryptoSignature,   // Signé par la clé privée du nœud
}
```

La chaîne cryptographique (chaque événement référence le hash du précédent) garantit qu'aucun événement ne peut être inséré ou supprimé rétrospectivement sans casser la chaîne — détectable immédiatement lors de la vérification.

12.2 Niveaux de Logging

Niveau	Contenu	Usage
AUDIT (obligatoire)	Toute action avec effet externe, tous les branchements de fallback, toutes les décisions LLM	Conformité réglementaire, responsabilité légale
EXECUTION	Début/fin de chaque instruction, durées, résultats	Debugging, performance monitoring, SLA
METRIC	Latences, compteurs, utilisation ressources	Dashboards opérationnels, alertes de performance
DEBUG	État interne SVM, décisions d'ordonnancement	Développement et investigation d'incidents
TRACE	Données complètes d'entrée/sortie	Investigation approfondie, jamais

Niveau	Contenu	Usage
	de chaque instruction	en production par défaut

12.3 Observabilité Distribuée

Dans une topologie multi-nœuds, l'observabilité est centralisée sur le nœud principal qui agrège les logs de tous les nœuds Rust via Kafka. Chaque nœud bufferise localement ses logs en cas de déconnexion.

Composant	Rôle
Kafka (topic: audit-events)	Transport des événements d'audit depuis tous les nœuds vers le nœud principal
Time-series DB (InfluxDB/TimescaleDB)	Stockage des métriques de performance pour dashboards
Audit DB (PostgreSQL append-only)	Stockage permanent des événements d'audit cryptographiques
Log aggregator (Loki / Elasticsearch)	Indexation des logs pour recherche et investigation
Dashboard (Grafana)	Visualisation temps réel de l'état de tous les nœuds et workflows

12.4 Requêtes d'Audit

Le système expose une API d'audit permettant de répondre à toute question réglementaire ou d'investigation :

```
"Qu'est-il exactement arrivé au patient 47 le 15/01 à 03:47:23 ?"
 → query(patient_id=47, timestamp="2025-01-15T03:47:23Z")
 → Retourne: événement déclencheur, version workflow utilisée,
             actions exécutées, durées, résultats, validateur du workflow

"La vanne V3 a-t-elle été fermée par le système aujourd'hui ?"
 → query(action=CLOSE_VALVE, target=V3, date=today)
 → Retourne: oui/non, timestamp précis, workflow décideur,
             valeur capteur qui a déclenché, postcondition vérifiée
```

13. Sécurité et Certifiabilité

13.1 Modèle de Sécurité

La sécurité du Semantic Compiler Platform est conçue selon le principe de "sécurité par construction" — les propriétés de sécurité sont garanties par l'architecture, pas par des garde-fous applicatifs qui peuvent être contournés.

Menace	Protection classique (n8n/agents)	Protection SCP
Injection de prompt	Filtres textuels, garde-fous LLM	Impossible : SVM exécute du bytecode, pas du texte
Action non autorisée	Vérification de permissions runtime	Actions définies à la compilation, catalogue signé
Exfiltration de données	Monitoring des accès	Connecteurs de sortie définis et validés à la compilation
Comportement imprévu	Monitoring comportemental	Structurellement impossible : programme figé
Secrets exposés	Variables d'environnement chiffrées	Vault injection runtime, jamais dans le LLM-IR
LLM-IR falsifié	Intégrité applicative	Signature cryptographique, refus si invalide

13.2 Gestion des Secrets

Les secrets (clés API, mots de passe, tokens) ne sont jamais inclus dans le LLM-IR ni exposés au compilateur LLM. Ils sont injectés par le vault au moment de l'exécution dans des slots pré-définis à la compilation.

```
// Dans le LLM-IR (sans secret)
LLM_CALL_INSTRUCTION {
  ...
  dynamic_slots: [
    { slot_id: "api_key", source: VaultSecret("sap/api_key") },
    { slot_id: "user_data", source: RuntimeData("event.payload") }
  ]
}
// Au runtime, la SVM demande au Vault la valeur de "sap/api_key"
// Le secret est utilisé pour l'appel et immédiatement détruit de la mémoire
```

13.3 Certifications Cibles par Secteur

Secteur	Certifications applicables
Médical (Europe)	MDR 2017/745, IEC 62304 (logiciel dispositif médical), HL7 FHIR
Médical (USA)	FDA 21 CFR Part 11, FDA Software as Medical Device (SaMD)
Industrie critique	IEC 61508 (sécurité fonctionnelle), SIL 1-3
Finance (Europe)	DORA (Digital Operational Resilience Act), PCI-DSS
Données personnelles	RGPD, audit trail pour droit à la suppression
Cybersécurité	IEC 62443 (sécurité systèmes industriels), ISO 27001

14. Marchés Adressables

14.1 Vue d'Ensemble du Marché

Le Semantic Compiler Platform adresse un marché total adressable (TAM) estimé à plus de 500 milliards d'euros, réparti entre les marchés critiques, industriels, métier et grand public. La stratégie de conquête recommande une approche séquentielle par verticale.

14.2 Marchés Critiques — Haute Valeur, Barrières Fortes

Marché	Taille estimée	Ticket moyen
Santé / Hôpitaux	50Md€ Europe	50k-500k€ / établissement
Énergie / Utilities	200Md€ mondial	200k-2M€ / installation
Transport / Infrastructure	100Md€ Europe	500k-5M€ / réseau
Défense / Sécurité	50Md€ Europe	1M-10M€ / déploiement

14.3 Marchés Industriels — Volume Important

Marché	Taille estimée	Avantage clé
Manufacturing / Industrie 4.0	150Md€ Europe	Remplacement PLCs, déterminisme certifié
Agroalimentaire	30Md€ Europe	Traçabilité obligatoire, chaîne du froid
Logistique / Entrepôts	50Md€ Europe	Automatisation workflows + IoT
Smart Buildings	40Md€ Europe	CVC, énergie, sécurité unifié

14.4 Agriculture — Impact Social Fort

L'agriculture de précision est le marché d'entrée recommandé pour sa rapidité de cycle de vente, l'accessibilité des décideurs, et les subventions disponibles (Plan France 2030, PAC européenne).

- Grandes cultures : irrigation, fertilisation, monitoring sols
- Élevage : surveillance individuelle, alimentation automatisée, détection maladies
- Serres et cultures protégées : contrôle climatique, optimisation production
- Viticulture : surveillance micro-climatique, traitements, prévision récoltes

14.5 Marchés Métier et Grand Public

Marché	Modèle économique recommandé
Finance / Banques	Licence enterprise + certification PCI-DSS/DORA
Télécommunications	SaaS B2B, ticket moyen, volume
PME tous secteurs	SaaS B2B, abonnement mensuel, self-service
Développeurs / DevOps	Freemium + SaaS, product-led growth
Grand public tech	SaaS B2C, abonnement mensuel faible

14.6 Stratégie de Conquête en 4 Temps

15. Temps 1 (0-18 mois) : Agriculture de précision en France. Un seul cas d'usage, un seul pays. Référence client, validation produit, subventions.
16. Temps 2 (18-36 mois) : Industrie agroalimentaire et manufacturing. Mêmes connecteurs, catalogue enrichi, clients adjacents naturels.
17. Temps 3 (36-60 mois) : Santé et énergie. Démarrer les certifications MDR/IEC 61508 dès le Temps 2. Tickets élevés, barrières à l'entrée.
18. Temps 4 (60 mois+) : Plateforme ouverte. Ouvrir le catalogue aux partenaires intégrateurs. Modèle marketplace. Effet de réseau.

15. Feuille de Route d'Implémentation

Phase 1 — Fondations (Mois 1-3)

Objectif : SVM fonctionnelle capable d'exécuter un DAG simple sur un cas d'usage réel.

- Définir le schéma Protocol Buffers du LLM-IR v1.0
- Implémenter la SVM Rust basique : chargement LLM-IR, exécution séquentielle, audit log
- Développer 3 connecteurs de base : Kafka consumer, HTTP call, Write file
- Implémenter le nœud NestJS principal avec API de compilation
- Créer le premier catalogue minimal (5-10 entrées) pour un cas IoT agricole
- Déployer sur Raspberry Pi 4 et valider le workflow de bout en bout

Phase 2 — Compilation Sémantique (Mois 3-6)

Objectif : LLM capable de générer un DAG valide depuis du langage naturel.

- Intégrer la génération contrainte (Outlines ou Guidance) pour forcer la grammaire LLM-IR
- Développer le DAG visualizer pour la validation humaine
- Implémenter le vérificateur formel léger (règles de types + terminaison)
- Construire le système de signature et versioning du LLM-IR
- Enrichir le catalogue : 20-30 connecteurs couvrant l'agriculture et l'IoT
- Première intégration Z3 pour vérification des postconditions critiques

Phase 3 — Execution Distribuée (Mois 6-9)

Objectif : Déploiement multi-nœuds fiable avec résilience.

- Implémenter le protocole de distribution LLM-IR via WebSocket TLS
- Développer le mode hors-ligne avec bufferisation locale et synchronisation
- Implémenter l'ordonnanceur parallèle dans la SVM
- Moteur de fallback complet avec les 5 stratégies
- Cross-compilation automatisée pour ARM Linux embarqué
- Dashboard d'observabilité (Grafana + Kafka audit events)

Phase 4 — LLM Calls Avancés (Mois 9-12)

Objectif : Orchestration multi-LLM et boucles de raisonnement.

- Système de construction statique de contexte LLM à la compilation
- Pipeline multi-LLM avec validation typée entre les nœuds
- Boucles de raisonnement bornées avec enrichissement de contexte
- Intégration de modèles locaux (Llama, Mistral) pour nœuds Linux embarqués
- Gestion des secrets via Vault avec injection runtime

Phase 5 — Production et Certifications (Mois 12-18)

Objectif : Système certifiable, premier client production, démarrage certifications.

- Audit trail cryptographique complet avec chaîne de hachage
- Support microcontrôleurs Rust no_std (STM32, ESP32)
- Démarrer le dossier de certification MDR pour usage médical
- Démarrer la certification IEC 61508 pour usage industriel critique
- Premier déploiement client production (agriculture ou industrie)
- Documentation complète API et catalogue pour partenaires intégrateurs

16. Ressources Techniques de Référence

16.1 Livres Fondamentaux

Théorie des Compilateurs

- "Crafting Interpreters" — Robert Nystrom (gratuit en ligne) : implémentation pratique d'un runtime
- "Engineering a Compiler" — Cooper & Torczon : IR et optimisations
- "Compilers: Principles, Techniques, and Tools" — Aho et al. : théorie complète
- "Types and Programming Languages" — Benjamin Pierce : systèmes de types formels

Systèmes Distribués et Streaming

- "Designing Data-Intensive Applications" — Martin Kleppmann : Kafka, cohérence, distribution
- "Kafka: The Definitive Guide" — Narkhede et al. : Kafka en production

Rust et Systèmes

- "Programming Rust" — Blandy et al. : ownership, async/await, Tokio
- "Rust for Rustaceans" — Jon Gjengset : niveau avancé, FFI, unsafe

Machines Virtuelles

- "Virtual Machines" — Smith & Nair : conception de jeux d'instructions et runtimes

16.2 Papers Académiques Essentiels

- "DSPy: Compiling Declarative Language Model Calls" — Khattab et al. (2023) : le plus proche de SCP
- "Neurosymbolic AI: The 3rd Wave" — Garcez & Lamb (2020) : fondations théoriques
- "Constrained Language Models Yield Few-Shot Semantic Parsers" — Shin et al. (2021) : génération contrainte
- "Program Synthesis with Large Language Models" — Austin et al. (2021, Google Brain)
- "Tree of Thoughts" — Yao et al. (2023) : raisonnement structuré en arbres
- "ReAct: Synergizing Reasoning and Acting in Language Models" — Yao et al. (2022)

16.3 Librairies et Outils Clés

Outil	Usage dans SCP
Outlines (Python)	Génération contrainte LLM pour compiler vers grammaire LLM-IR
Z3 SMT Solver (Rust: z3 crate)	Vérification formelle des propriétés du LLM-IR
Embassy (Rust)	Async runtime pour microcontrôleurs Rust no_std
Tokio (Rust)	Async runtime pour nœuds Rust Linux
Protocol Buffers (prost crate)	Sérialisation binaire du LLM-IR
LLVM IR Spec (référence)	Inspiration pour la conception du format LLM-IR
probe-rs	Flashing et debugging des microcontrôleurs Rust
BullMQ (NestJS)	Queue de jobs pour la distribution des LLM-IR