SEMANTIC COMPILER PLATFORM

Complete Technical Specification

Architecture · Features · Implementation · Deployment

Universal Deterministic Automation Infrastructure

Version 1.0 — 2025

1. Vision and Strategic Positioning

1.1 The Fundamental Problem

Current automation systems force a choice between two incompatible approaches. On the one hand, LLM agents (LangGraph, n8n with AI, OpenClaw) are flexible and capable of reasoning about unforeseen cases, but their behavior is probabilistic, non-certifiable, and potentially dangerous in critical contexts. On the other hand, traditional industrial automation systems (PLCs, RPA) are deterministic and certifiable, but rigid, costly to program, and inaccessible to non-technical users.

This tension creates a gaping hole in the market: entire sectors—medical, industrial, agricultural, critical infrastructure—cannot adopt AI because they cannot afford non-determinism, but also cannot bear the rigidity and cost of conventional systems.

The Core Value Proposition

The Semantic Compiler Platform resolves this tension by using the LLM as a static compiler, not as a dynamic decision maker. Intelligence is confined to the compilation phase, supervised by humans, and formally verified. Execution is purely deterministic, fast, and certifiable.

1.2 The Fundamental Distinction

Dimension	Agent systems (LangGraph, n8n)	Semantic Compiler Platform
LLM at runtime	Systematic at every stage	Zero — confined to compilation
Determinism	Probabilistic, variable	Absolute by construction
Certifiability	Impossible	IEC 61508, MDR, FDA possible
Edge deployment	Minimum Linux server	Microcontroller 64KB RAM
Human validation	None	DAG validated before execution
Runtime hallucinations	Possible	Structurally impossible
Audit trail	Application logs	Cryptographically signed
Target user	Python/JS developers	Non-technical professions

1.3 Unique Market Position

The Semantic Compiler Platform creates a new category that no competitor can address with its current architecture:

- n8n / Make / Zapier: cloud SaaS automation, no edge deployment, no certifiability, non-deterministic LLM

- LangGraph / LlamaIndex: frameworks for developers, LLM decision-maker at runtime, probabilistic behavior
- Industrial PLCs (Siemens, Rockwell): deterministic but specialized programming, high cost, no natural language
- OpenClaw: agentic personal assistant, non-deterministic, not certifiable, no embedded deployment

Unique Positioning

"The only system that allows a non-technical operator to describe a critical workflow in natural language, validate it visually, and execute it identically on an industrial microcontroller or a cloud server—with the same guarantee of determinism, security, and auditability."

2. Overall System Architecture

2.1 Overview

The system is structured around two distinct phases and a distributed runtime. This separation is the fundamental architectural guarantee of determinism.

```
┌─────────────────────────────────────────────────────────────────┐
│                    COMPILATION PHASE (offline)                │
│                                                                 │
│  Natural language → LLM Parser → Semantic DAG                 │
│                          ↓                                     │
│                  Human validation                             │
│                          ↓                                     │
│              Formal verifier (SMT/Z3)                       │
│                          ↓                                     │
│              Signed binary LLM-IR + versioned                  │
└─────────────────────────┬───────────────────────────────────────┘
                          │
┌─────────────────────────▼───────────────────────────────────────┐
│                   MAIN NODE (Nest.js)                       │
│              ws://server:18789 — Control plane                │
│
│  Orchestration · Catalog · Sessions · Distribution · Audit    │
└────────────┬────────────────────────────────────┬───────────────┘
             │  Signed LLM-IR                      │  Signed LLM-IR
             ▼                                    ▼
┌────────────────────────┐           ┌─────────────────────────────┐
│  RUST NODE — Linux     │           │  RUST NODE — Embedded       │
│  (Raspberry Pi, etc.)  │           │  (STM32, ESP32, etc.)       │
│                        │           │                             │
│  Kafka · MQTT · HTTP   │           │  no_std · Embassy async     │
│  Complex workflows   │           │  Sensors · Actuators    │
│  Bounded LLM calls       │           │  Minimal LLM-IR             │
└────────────────────────┘           └─────────────────────────────┘
```

2.2 The Two Brains of the System

The system implements a "neurosymbolic" architecture with a formal separation between two processing modes:

Mode	Description
Deterministic brain (SVM)	For known and routine cases. Purely mechanical execution of the compiled LLM-IR. Zero LLM. Latency 10-50ms. Absolute guarantee of identical behavior.
Reasoning brain (bounded LLM)	For unforeseen cases not covered by compiled fallbacks. LLM called with pre-built bounded context, output formally validated before execution. Max 3 attempts then human escalation.

2.3 Technology Stack

Component	Technology	Justification
Main node	NestJS (Node.js ≥22)	Rich ecosystem, DI, WebSocket, native BullMQ, rapid development
Execution nodes	Rust (stable)	No GC, memory safety, standalone binary, embedded possible
LLM-IR format	Protocol Buffers (protobuf)	Typed, versioned, compact, cross-language binary
Event bus	Apache Kafka	Persistence, replay, exactly-once, producer/consumer decoupling
Embedded async	Embassy (Rust)	async/await on microcontroller, no RTOS
Formal verification	Z3 SMT Solver (Rust binding)	Formal proof of termination and constraints
LLM-IR transport	WebSocket + TLS	Automatic reconnection, low overhead
Audit log	Append-only + SHA-256	Immutable, regulatory certifiable
Secret management	HashiCorp Vault / runtime injection	Secrets never exposed to the compiler or stored in LLM-IR

3. Semantic Compilation — The Heart of the System

3.1 Compilation Philosophy

Semantic compilation is the process that transforms an intention expressed in natural language into a validated deterministic binary program. It operates in several successive phases, each progressively reducing ambiguity and increasing the level of formal guarantee.

Fundamental Principle

The LLM never makes decisions during execution. It makes proposals during compilation. The machine decides whether the proposal is formally valid. The human validates what will be done. This triptych — LLM proposes, formal validates, human approves — is the guarantee against hallucinations with real impact.

3.2 Compilation Phases

1. Phase 1: Linguistic Analysis (LLM Parser)

The LLM Parser receives the natural language query and produces a structured intermediate representation. At this stage, generation is constrained by the grammar of the available catalog — the LLM cannot generate instructions referencing non-existent connectors.

```
Input: "If the pressure exceeds 8.5 bar, close valve V3 and alert."
Output: Structured AST {trigger: pressure_sensor, condition: "> 8.5 bar",
          actions: [close_valve_V3, alert_operator],
          type: event_driven, criticality: high}
```

2. Phase 2: Catalog Resolution

The compiler maps the elements of the AST to the concrete, typed entries in the catalog. Each connector, skill, and service is resolved with its complete metadata: preconditions, postconditions, security constraints, estimated latency.

3. Phase 3: Semantic DAG Construction

The Directed Acyclic Graph is constructed with all nominal paths AND fallback branches. The user is prompted to select their strategy for each potential contingency identified during compilation.

4. Phase 4: Human Validation

The DAG is presented to the user in a visual and readable form. They see exactly what the system will do, in what order, with what data, and what strategy applies in the event of an anomaly. They can modify, enrich, or reject it.

5. Phase 5: Formal Verification

Once validated by a human, the DAG goes through the formal verifier (Z3 SMT Solver). This verifier guarantees: program termination (no infinite loops), compliance with resource constraints, consistency of types between nodes, and satisfaction of preconditions/postconditions.

6. Phase 6: LLM-IR Generation and Signature

The validated and verified DAG is compiled into binary LLM-IR in Protocol Buffers format. This binary is cryptographically signed (SHA-256 + instance private key) and versioned. The Rust node refuses to execute any LLM-IR that is not signed or whose signature does not match.

3.3 Constrained Generation

To ensure that there are no hallucinations during compilation, the LLM operates in constrained generation mode. Invalid tokens—i.e., references to connectors or actions not present in the catalog—are masked during generation. The LLM cannot physically generate a program that references a non-existent resource.

```
// The LLM-IR grammar is defined by the catalog
// Only valid tokens can be generated at each position
allowed_tokens_at_position = catalog.valid_next_tokens(current_context)
// Tokens not in the catalog are masked with logit_bias = -100
```

3.4 The LLM Context Built at Compile Time

Unlike traditional systems that build the LLM context at runtime (the main source of hallucinations), the compiler statically builds the optimal context for each LLM_CALL node in the DAG. This context is frozen in the LLM-IR.

```
struct CompiledLLMContext {
  system_prompt: String,          // frozen at compilation
  few_shot_examples: Vec<Example>,// selected at compilation
  output_schema: JsonSchema,      // formal validation of the output
  model: ModelIdentifier,         // optimal model chosen at compilation
  temperature: f32,              // calibrated according to node requirements
  max_tokens: u32,               // formally bounded
  dynamic_slots: Vec<ContextSlot>,// slots for runtime data
}
```

At runtime, the SVM only injects dynamic data into predefined slots. The LLM receives a perfectly calibrated context, which drastically reduces the risk of hallucination.

3.5 Controlled Reasoning Loops

For problems requiring iterative reasoning (diagnostics, complex analysis), the compiler generates bounded reasoning loops. Unlike classic agent loops, which can diverge indefinitely, these loops have formal exit conditions compiled into the LLM-IR.

```
struct LoopInstruction {
  max_iterations: u8,             // never > 5 by default
  timeout_ms: u32,               // absolute timeout
  convergence_predicate: Predicate, // formal termination condition
  context_enrichment: EnrichmentStrategy, // how to enrich each round
  fallback: FallbackInstruction,  // if convergence not reached
}
```

4. The Catalog — System Trust Layer

4.1 Role and Philosophy

The catalog is the cornerstone of system security. It is a structured, signed, and versioned registry of everything the system can do. The LLM can only compose elements from the catalog — it cannot invent new capabilities. This constraint is the fundamental guarantee against hallucinations with real-world impact.

4.2 Structure of a Catalog Entry

```
interface CatalogEntry {
  // Identity
  id: string;              // Stable UUID
  name: string;            // Human-readable name
  version: SemVer;         // Strict versioning
  category: EntryCategory; // trigger | action | transform | llm_call | service

  // Formal contract
  input_schema: JsonSchema;   // Strict input types
  output_schema: JsonSchema;  // Strict output types
  preconditions: Predicate[]; // Conditions required before execution
  postconditions: Predicate[];// Guarantees after successful execution

  // Performance metadata
  estimated_latency_ms: Range; // Min/max latency
  resource_cost: ResourceCost; // CPU, RAM, network
  rate_limits: RateLimit[];    // Call limits

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

  // Signature
  author: string;
  signature: CryptoSignature; // Signed by the creator
}
```

4.3 Catalog Categories

Category	Examples	Characteristics
Triggers	Kafka consumer, FS watcher, IoT MQTT, Webhook, Cron, DB poll, WebSocket	Event sources. Trigger the execution of the compiled workflow.
Control actions	Close valve, Activate pump, Restart service, Send command	Act on the physical world. Maximum security constraints, time windows, rate limits.
Data actions	INSERT SQL, Publish Kafka, Write file, POST API, Send email/SMS	Data modifications. Mandatory verification postconditions.
Transformations	OCR, PDF parser, filtering, aggregation, format conversion, extraction	Pure processing. No side effects. Parallelizable.
LLM Calls	Claude Opus, GPT-4o, Specialized models, Local models	LLMs as bounded services. Pre-built context. Schematized and validated output.
Business services	SAP, Salesforce CRM, medical EHR, agricultural ERP, industrial SCADA	Connectors to enterprise systems. Authentication managed by Vault.
Readings	Read sensor, Query DB, Fetch API, Read file, Consume event	Data sources. Frequency, cache, defined timeouts.

4.4 Specialized Vertical Catalogs

The catalog is extensible by domain. Each vertical sector has its own specialized catalog, developed and certified by business experts:

- Medical Catalog: EHR/HIS connectors, MDR-certified patient sensors, FHIR protocols, medical actions with double validation
- Industrial Catalog: Modbus, OPC-UA, industrial MQTT protocols, IEC-certified sensors, actuators with SIL
- Agricultural Catalog: soil/weather sensors, irrigation systems, weather APIs, agricultural ERPs, CAP reports
- Finance Catalog: PCI-DSS banking connectors, trading APIs, DORA regulatory reporting systems
- Consumer IoT Catalog: home appliances, cloud APIs (Google, Amazon), mobile notifications

4.5 Signature and Governance System

Each catalog entry is cryptographically signed by its author. The governance system allows you to define who can add, modify, or revoke catalog entries depending on the deployment context.

Role	Permissions
System Administrator	Add/revoke entries in all catalogs
Certified developer	Add entries to catalogs in their domain
Business Operator	Use catalog entries to create workflows
Auditor	Read-only access to catalog history and signatures

5. LLM-IR — Deterministic Intermediate Format

5.1 Format Philosophy

LLM-IR (Large Language Model Intermediate Representation) is the binary format in which compiled DAGs are stored and distributed. It is inspired by the design of LLVM IR—a typed, versioned, and optimizable intermediate representation that decouples the intelligence phase from the execution phase.

Fundamental Properties of LLM-IR

• Typed: each instruction has strict input and output types, validated at compile time • Versioned: version incompatibility = execution refusal by the SVM • Signed: SHA-256 cryptographic signature, unalterable • Deterministic: same LLM-IR = same behavior, always • Compact: Protocol Buffers, approximately 2-10KB for a typical workflow

5.2 Structure of the LLM-IR

```
message LlmIR {
  string dag_id = 1;
  uint32 version = 2;
  bytes signature = 3;           // SHA-256 of the set
  string validated_by = 4;       // Validator user ID
  int64 compiled_at = 5;         // Compilation timestamp
  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;   // Versioned catalog reference
  bytes compiled_context = 4;    // Pre-built context (for 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 and Compatibility

LLM-IR uses strict semantic versioning. The SVM maintains a compatibility matrix:

Version change	SVM behavior
Same major version	Execution allowed with warning if minor version is different
Different major version	Execution denied, request for recompilation
Invalid signature	Absolute refusal, immediate security alert
Catalog entry revoked	Execution denied, notification to master node

5.4 DAG Storage Management

The system maintains two synchronized representations of the DAG, with a strict unidirectional derivation rule:

```
JSON DAG (human source of truth)
  → Stored in a versioned database (PostgreSQL)
  → Readable and editable by operators
  → Contains the corresponding LLM-IR hash
  ↓ (unidirectional derivation, never the reverse)
Binary LLM-IR (machine source of truth)
  → Stored as a signed artifact
  → Never edited directly
  → Distributed to Rust nodes via secure channel
```

6. The Semantic Virtual Machine (SVM) — High-Performance Runtime

6.1 Architecture of the Rust SVM

The SVM is the executive core of the system, implemented in Rust for maximum performance, memory safety, and determinism guarantees. It interprets the LLM-IR and orchestrates the execution of instructions without ever making autonomous decisions.

```
pub struct SVM {
  ir_loader: IrLoader,           // Loads and validates LLM-IR
  scheduler: Scheduler,          // Parallel/sequential scheduling
  memory: ExecutionMemory,       // Deterministic memory management
  connector_registry: Registry,  // Connectors loaded from catalog
  fallback_engine: FallbackEngine,// Management of unforeseen cases
  audit_writer: AuditWriter,     // Cryptographic append-only writing
  health_monitor: HealthMonitor, // Connection health monitoring
  secret_vault: VaultClient,     // Injection of secrets at runtime
}
```

6.2 The Scheduler

The scheduler analyzes the dependencies encoded in the LLM-IR to determine which instructions can be executed in parallel and which must be sequential. This analysis is static—determined at compile time, not at runtime.

```
// Example of safe parallel execution
// Actions with no mutual dependencies → automatically parallelized
PARALLEL {
  READ temperature_sensor     // no dependencies
  READ pressure_sensor        // no dependencies
  FETCH weather_api              // no dependencies
}
// Synchronization on the three results
EVALUATE_CONDITIONS(temperature, pressure, weather)
// Dependent actions → sequential
IF condition_met:
  CONTROL close_valve         // depends on evaluation
  WRITE audit_log              // depends on the control action
  ALERT operator              // depends on the log
```

6.3 Execution Memory Management

The execution memory is structured in three layers according to lifetime and persistence requirements:

Layer	Data	Persistence
Persistent state	Trigger counters, last execution, condition history	Append-only database, survives restarts
Buffer pipeline	Output Action N → Input Action N+1, intermediate results	Memory only, lifetime = one execution
Execution context	User ID, permissions, project ID	Never persisted, rehydrated from authoritative source on each execution

6.4 The Fallback Engine

The fallback engine handles situations where the nominal path cannot be followed. All fallback strategies are defined at compile time by the user—there are no autonomous decisions at runtime.

Strategy	Behavior
FAIL_SAFE	Immediate workflow shutdown, operator alert, wait for human intervention. For safety-critical systems.
DEGRADED_MODE	Execution of the safest pre-compiled alternative path. For non-critical business workflows.
RETRY_WITH_BACKOFF	N attempts with exponential delay before escalation. For transient failures (network, API).
LLM_REASONING	Activation of the bounded LLM reasoner with pre-built context. For unforeseen cases not covered.
SUPERVISED_RECOMPILE	Triggers a new compilation phase with notification to the user. For context evolution.

6.5 Concurrency Management on Shared Resources

When multiple compiled workflows require the same resource simultaneously, the SVM applies the priority policy defined at compile time—never a decision made at runtime.

```
// Priority policy compiled in LLM-IR
struct PriorityPolicy {
  priority_level: u8,        // 0 = critical, 255 = low priority
  preemptible: bool,         // can be interrupted by higher priority
  max_wait_ms: u32,          // max delay before fallback if resource is busy
  fallback: FallbackInstruction,
}
```

7. Event Sources and Triggers

7.1 Event-Driven Architecture

The system is fundamentally event-driven. Unlike tools such as n8n, which operate primarily in pull mode (periodic polling), SVM maintains persistent listeners for each type of event source, enabling instantaneous response.

Critical Advantage Over Polling

A polling-based system checks the status every N seconds—if an incident occurs between two checks, the response time can be fatal in a medical or industrial context. The SVM's event-driven listeners respond within milliseconds of the event.

7.2 Apache Kafka Integration

Kafka is the primary bus for high-frequency events and IoT data. The integration offers critical guarantees for reliable systems:

- Exactly-once delivery: each event is processed once and only once, even if the node crashes
- Event persistence: events are retained, allowing replay for auditing or debugging
- Offset management: the SVM picks up exactly where it left off after a restart
- Consumer groups: multiple nodes can consume the same topic in parallel for scalability

```
// Kafka configuration compiled in 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>,      // pre-filter before trigger
  exactly_once: bool,
}
```

7.3 Supported Event Sources

Source	Protocol/Tech	Typical Use Cases
Apache Kafka	Kafka Consumer API	High-frequency IoT, business

Source	Protocol/Tech	Typical Use Cases
		events, system logs, data streams
MQTT (IoT)	MQTT 3.1.1 / 5.0	Industrial sensors, connected devices, home automation
Modbus	Modbus TCP/RTU	Industrial controllers, PLCs, energy meters
OPC-UA	OPC Unified Architecture	High-reliability industrial equipment
File System	inotify (Linux), FSEvents (macOS)	Folder monitoring, incoming file processing
HTTP Webhooks	HTTP/HTTPS POST	External APIs, SaaS events, cloud integrations
WebSocket	WebSocket protocol	Real-time data streams, control interfaces
Cron / Schedule	Cron expression	Periodic tasks, scheduled reports, maintenance
Database CDC	Debezium / pg_logical	Database changes (insert, update, delete)
Email (IMAP/Gmail)	IMAP / Gmail PubSub	Automatic processing of emails, invoices, alerts
System signals	SIGTERM, SIGINT, custom	System events, maintenance triggers

8. Distributed Execution

8.1 Multi-Node Topology

The system supports a flexible distributed topology adapted to each deployment context. The main NestJS node orchestrates a set of Rust execution nodes that can be deployed on a variety of hardware.

```
Typical topology for a smart factory:

Central server (Nest.js) — Cloud or on-premise
         │
         ├── Rust node #1: Raspberry Pi 4 — Production line A
         │         └── STM32 sub-nodes: temperature/pressure sensors
         │
         ├── Rust node #2: Industrial PC — Control room
         │         └── SCADA/OPC-UA connection
         │
         ├── Rust node #3: Raspberry Pi — Production line B
         │
         └── Rust Node #4: ESP32 — Environmental Sensors
```

8.2 Main Node ↔ Rust Nodes Communication Protocol

The communication protocol is designed for resilience — nodes continue to function even if the connection to the central server is temporarily interrupted.

Channel	Usage
WebSocket TLS (persistent)	Distribution of compiled LLM-IRs, catalog updates, management commands
HTTP REST (one-shot)	Status requests, health checks, log requests
Kafka (events)	Reporting of execution results, metrics, audit events

8.3 Resilience and Offline Mode

Each Rust node operates autonomously once the LLM-IR has been received. In the event of a loss of connection with the central server:

- The node continues to execute locally stored compiled workflows
- Execution results and audit events are buffered in local flash memory

- Upon reconnection, the node automatically synchronizes its status with the central server
- LLM_CALLs requiring external access are put on hold or trigger the FAIL_SAFE fallback depending on the compiled configuration

8.4 Deployment on Connected Objects

The Rust SVM is designed to be deployed on a wide range of hardware, from servers to microcontrollers:

Hardware Category	Examples	SVM Capabilities
Servers / VMs	Linux x86_64, ARM64 servers	Full SVM, all connectors, local or cloud LLM_CALL
Embedded Linux	Raspberry Pi 4, NVIDIA Jetson, Siemens IOT2050	Full SVM, Kafka, MQTT, HTTP, cloud LLM_CALL
ARM microprocessors	Raspberry Pi Zero 2W, BeagleBone	Standard SVM, lightweight connectors, local buffer
ARM Cortex-M microcontrollers	STM32, nRF52, SAMD51	SVM no_std, I2C/SPI/UART connectors, Embassy async
WiFi/BLE microcontrollers	ESP32, ESP32-C3 (RISC-V)	SVM no_std, WiFi/MQTT, basic IoT connectors

```
# Cross-compilation from development machine to embedded target
# 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
# Standalone binary, ~200KB, no runtime dependencies
```

9. Physical Control — Actions on the Real World

9.1 Types of Control Actions

The system supports three categories of physical actions, each with security assurance levels adapted to their criticality:

Type	Examples	Security Guarantees
Direct control	Close valve, activate pump, adjust temperature, start motor	Time windows, rate limits, postconditions, cancellation window
System commands	Restart service, Trigger emergency procedure, Change machine status	Compiled authorizations, mandatory audit, human confirmation if configured
IoT actuators	Relays, servomotors, electric valves, regulators	Compiled physical constraints (min/max), protection against invalid commands

9.2 Security Mechanisms for Physical Control

Authorization Time Windows

Each control action can be limited to specific time ranges, defined at compile time and validated by the user. The SVM systematically checks the window before any execution.

```
struct TimeWindow {
  days: Vec<Weekday>,      // authorized days
  start: NaiveTime,        // start time
  end: NaiveTime,          // end time
  timezone: Tz,            // explicit timezone
}
// If outside window → compiled fallback, NEVER perform action
```

Cancellation window for irreversible actions

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

Mandatory postcondition verification

After each physical control action, the SVM verifies that the actual state of the system matches the expected state (postcondition). If the verification fails, the compiled fallback is triggered.

9.3 Complete Example: Hybrid Industrial Workflow

This workflow crosses the physical/digital boundary in the same compiled DAG—a capability unique to the Semantic Compiler Platform:

```
DETECT machine anomaly (Kafka IoT sensor)
     ↓
CREATE automatic ticket (Jira REST API)
     ↓
NOTIFY technician (Slack + SMS)
     ↓
IF no response within 15 minutes:
  REDUCE machine speed (Modbus actuator) ← physical world
  [postcondition: measured_speed < target_speed + 5%]
     ↓
ORDER spare part (SAP ERP)
     ↓
UPDATE production schedule (Excel/SAP)
     ↓
NOTIFY affected customer (automatic email)
```

10. Advanced LLM Calls — Limited Intelligence

10.1 LLM as a Deterministic Service

In the Semantic Compiler Platform, LLMs are not decision-makers — they are specialized semantic processors, called as services with typed inputs and formally validated outputs. This distinction is fundamental.

```
// Classic LLM (agent) — dangerous
LLM receives vague context → decides what to do → executes

// LLM in SCP — controlled
SVM executes instruction LLM_CALL {
  inject(dynamic_data, pre_built_context) → calls LLM
  validate_output(output, output_schema)
  if invalid → retry(max=3) → fallback
  else → pass_typed_output_to_next_node()
}
```

10.2 Static Construction of the LLM Context

The compiler analyzes each LLM_CALL node in the DAG and constructs the optimal context at compile time, not at runtime. This pre-constructed context is stored in the LLM-IR and injected directly by the SVM during the call.

Why this is revolutionary

An LLM with a precise, bounded, and calibrated context hallucinates infinitely less than an LLM with a context built on the fly. The compiler can select the most relevant few-shot examples, calibrate the temperature as needed (extraction = 0.0, reasoning = 0.3), and define exactly the expected output format. All this is done once, at cold start, by a system that has the time to do things right.

10.3 Multi-LLM Orchestration

The compiler can generate DAGs that orchestrate several different LLMs in a pipeline, each receiving exactly the context it needs and producing a typed output consumed by the next one:

```
// Multi-LLM pipeline compiled for medical analysis
READ imaging_report (DICOM)
     ↓
LLM_CALL_1: Claude Opus
  context: [radiology_specialty, annotated_examples, structured_format]
  output_schema: {anomalies: [], confidence: float, zones: []}
     ↓ validated output
LLM_CALL_2: GPT-4o Vision (if confidence < 0.85)
  context: [previous_analysis, zoomed_suspicious_areas, guidelines]
  output_schema: {validation: bool, confidence: float, notes: string}
     ↓ validated output
LLM_CALL_3: Specialized medical model
  context: [results_1_2, patient_history, facility_protocols]
  output_schema: structured_medical_report
     ↓
WRITE medical_record (EHR)
ALERT responsible_physician (if critical anomaly)
```

10.4 Iterative Reasoning Loops

For problems requiring iterative reasoning (diagnosis, complex problem solving), the compiler generates bounded loops where each iteration enriches the context with previous results.

```
LOOP fault_diagnosis (max_iterations=3, timeout=30s) {
  LLM_CALL: DiagnosticModel
    input: {sensor_data, fault_history, previous_iteration}
    output: {probable_cause, confidence, missing_data[]}

  IF confidence > 0.9 → EXIT (success)
  IF missing_data → COLLECT missing_data → CONTINUE
  IF iteration >= max → FALLBACK: human_escalation
}
```

11. Versioning, Modifiability, and Lifecycle Management

11.1 Versioning System

Versioning is the mechanism that guarantees complete traceability of workflow changes. It is designed to meet a fundamental regulatory requirement: to be able to prove, for any past execution, exactly which program was used to perform it and who validated it.

```
interface WorkflowVersion {
  task_id: UUID;          // Stable — workflow identity
  version: number;        // Incremental — program identity
  parent_version: number; // Traceability of evolution
  status: "active" | "archived" | "executing" | "deprecated";
  dag_json: object;       // Human source of truth
  ir_checksum: string;    // SHA-256 hash of the corresponding LLM-IR
  validated_by: string;   // Validator user ID
  validated_at: DateTime; // Validation timestamp
  change_reason: string;  // Reason for change
}
```

11.2 Change Lifecycle

7. Change request: the user initiates a change to the active workflow
8. Status check: the SVM checks whether any executions are in progress on the active version
9. Management of active executions: wait for completion or force a clean shutdown with audit
10. New compilation: version N+1 is compiled from the modified DAG
11. Human validation: the new DAG is presented for validation
12. Formal verification: Z3 verifies the properties of the new program
13. Signing and deployment: the new LLM-IR is signed and distributed to the nodes
14. Archiving: version N is archived — immutable, never deleted

11.3 Archiving and Audit Policy

No version is ever deleted from the system. This rule is fundamental to regulatory compliance. The complete execution history is retained with reference to the exact version used.

Data type	Retention policy
Workflow versions (DAG JSON)	Permanent retention, immutable after archiving
Signed LLM-IR binaries	Permanent retention, unalterable
Execution logs	Minimum 10 years for critical medical/industrial use
Cryptographic audit trail	Permanent storage, append-only
Execution results	Configurable depending on context (30 days to permanent)

12. Logging, Audit, and Observability

12.1 The Cryptographic Audit Trail

The audit trail is the system's legal and regulatory accountability layer. Each significant event is recorded in append-only mode and cryptographically signed, ensuring that no entry can be modified or deleted after the fact.

```
struct AuditEvent {
  event_id: UUID,
  timestamp: DateTime<Utc>,     // Nanosecond precision
  node_id: String,              // Node that generated the event
  workflow_id: UUID,
  workflow_version: u32,        // EXACT version used
  instruction_id: String,       // Precise instruction in the LLM-IR
  event_type: AuditEventType,   // EXECUTION_START | ACTION_TAKEN | FALLBACK | ...
  input_hash: [u8; 32],         // Hash of input data
  output_hash: [u8; 32],        // Hash of output data
  duration_ms: u64,
  result: ExecutionResult,
  previous_event_hash: [u8; 32],// Cryptographic string (blockchain-like)
  signature: CryptoSignature,   // Signed by the node's private key
}
```

The cryptographic chain (each event references the hash of the previous one) ensures that no event can be inserted or deleted retrospectively without breaking the chain — which is immediately detectable during verification.

12.2 Logging Levels

Level	Content	Usage
AUDIT (mandatory)	Any action with external effect, all fallback connections, all LLM decisions	Regulatory compliance, legal liability
EXECUTION	Start/end of each instruction, durations, results	Debugging, performance monitoring, SLA
METRIC	Latencies, counters, resource usage	Operational dashboards, performance alerts
DEBUG	Internal SVM status, scheduling decisions	Development and incident investigation
TRACE	Complete input/output data for each	In-depth investigation, never in

Level	Content	Usage
	instruction	production by default

12.3 Distributed Observability

In a multi-node topology, observability is centralized on the main node, which aggregates logs from all Rust nodes via Kafka. Each node buffers its logs locally in case of disconnection.

Component	Role
Kafka (topic: audit-events)	Transport of audit events from all nodes to the main node
Time-series DB (InfluxDB/TimescaleDB)	Storage of performance metrics for dashboards
Audit DB (PostgreSQL append-only)	Permanent storage of cryptographic audit events
Log aggregator (Loki/Elasticsearch)	Log indexing for search and investigation
Dashboard (Grafana)	Real-time visualization of the status of all nodes and workflows

12.4 Audit requests

The system provides an audit API that can be used to answer any regulatory or investigative questions:

```
"What exactly happened to patient 47 on 01/15 at 03:47:23?"
 → query(patient_id=47, timestamp="2025-01-15T03:47:23Z")
 → Returns: triggering event, workflow version used,
             actions performed, durations, results, workflow validator

"Was valve V3 closed by the system today?"
 → query(action=CLOSE_VALVE, target=V3, date=today)
 → Returns: yes/no, precise timestamp, decision-making workflow,
             triggering sensor value, verified postcondition
```

13. Security and Certifiability

13.1 Security Model

The security of the Semantic Compiler Platform is designed according to the principle of "security by construction" — security properties are guaranteed by the architecture, not by application safeguards that can be circumvented.

Threat	Classic protection (n8n/agents)	SCP protection
Prompt injection	Text filters, LLM safeguards	Not possible: SVM executes bytecode, not text
Action not allowed	Runtime permission verification	Actions defined at compile time, signed catalog
Data exfiltration	Access monitoring	Output connectors defined and validated at compile time
Unexpected behavior	Behavioral monitoring	Structurally impossible: frozen program
Exposed secrets	Encrypted environment variables	Vault injection runtime, never in the LLM-IR
Falsified LLM-IR	Application integrity	Cryptographic signature, rejection if invalid

13.2 Secret Management

Secrets (API keys, passwords, tokens) are never included in the LLM-IR or exposed to the LLM compiler. They are injected by the vault at runtime into slots predefined at compile time.

```
// In the LLM-IR (without secrets)
LLM_CALL_INSTRUCTION {
  ...
  dynamic_slots: [
    { slot_id: "api_key", source: VaultSecret("sap/api_key") },
    { slot_id: "user_data", source: RuntimeData("event.payload") }
  ]
}
// At runtime, the SVM requests the value of "sap/api_key" from the Vault
// The secret is used for the call and immediately destroyed from memory
```

13.3 Target Certifications by Sector

Sector	Applicable Certifications
Medical (Europe)	MDR 2017/745, IEC 62304 (medical device software), HL7 FHIR
Medical (USA)	FDA 21 CFR Part 11, FDA Software as Medical Device (SaMD)
Critical industry	IEC 61508 (functional safety), SIL 1-3
Finance (Europe)	DORA (Digital Operational Resilience Act), PCI-DSS
Personal data	GDPR, audit trail for right to erasure
Cybersecurity	IEC 62443 (industrial systems security), ISO 27001

14. Addressable Markets

14.1 Market Overview

The Semantic Compiler Platform addresses a total addressable market (TAM) estimated at over €500 billion, spread across critical, industrial, business, and consumer markets. The market entry strategy recommends a sequential approach by vertical.

14.2 Critical Markets — High Value, Strong Barriers

Market	Estimated Size	Average Ticket
Healthcare/Hospitals	€50 billion Europe	€50k-500k / facility
Energy/Utilities	€200 billion worldwide	€200k-€2M / facility
Transportation/ Infrastructure	€100 billion in Europe	€500k-5M / network
Defense/Security	€50 billion Europe	€1 million-€10 million / deployment

14.3 Industrial Markets — Large Volume

Market	Estimated size	Key advantage
Manufacturing / Industry 4.0	€150 billion Europe	Replacement of PLCs, certified determinism
Agri-food	€30 billion Europe	Mandatory traceability, cold chain
Logistics/Warehouses	€50 billion Europe	Workflow automation + IoT
Smart buildings	€40 billion Europe	HVAC, energy, unified security

14.4 Agriculture — Strong Social Impact

Precision agriculture is the recommended entry market due to its fast sales cycle, accessibility of decision-makers, and available subsidies (France 2030 Plan, European CAP).

- Field crops: irrigation, fertilization, soil monitoring
- Livestock: individual monitoring, automated feeding, disease detection
- Greenhouses and protected crops: climate control, production optimization
- Viticulture: microclimate monitoring, treatments, harvest forecasting

14.5 Professional and consumer markets

Market	Recommended business model
Finance/Banks	Enterprise license + PCI-DSS/DORA certification
Telecommunications	B2B SaaS, average ticket, volume
SMEs in all sectors	B2B SaaS, monthly subscription, self-service
Developers/DevOps	Freemium + SaaS, product-led growth
Consumer tech	B2C SaaS, low monthly subscription

14.6 Four-step conquest strategy

15. Phase 1 (0-18 months): Precision agriculture in France. Single use case, single country. Customer references, product validation, subsidies.
16. Phase 2 (18-36 months): Agri-food and manufacturing. Same connectors, expanded catalog, natural adjacent customers.
17. Phase 3 (36-60 months): Healthcare and energy. Begin MDR/IEC 61508 certifications in Phase 2. High ticket prices, barriers to entry.
18. Phase 4 (60 months+): Open platform. Open the catalog to integration partners. Marketplace model. Network effect.

15. Implementation Roadmap

Phase 1 — Foundations (Months 1-3)

Objective: Functional SVM capable of executing a simple DAG on a real use case.

- Define the Protocol Buffers schema for LLM-IR v1.0
- Implement basic Rust SVM: LLM-IR loading, sequential execution, audit log
- Develop 3 basic connectors: Kafka consumer, HTTP call, Write file
- Implement the main NestJS node with compilation API
- Create the first minimal catalog (5-10 entries) for an agricultural IoT use case
- Deploy on Raspberry Pi 4 and validate the end-to-end workflow

Phase 2 — Semantic Compilation (Months 3-6)

Objective: LLM capable of generating a valid DAG from natural language.

- Integrate constraint generation (Outlines or Guidance) to enforce LLM-IR grammar
- Develop the DAG visualizer for human validation
- Implement the lightweight formal verifier (type rules + termination)
- Build the LLM-IR signature and versioning system
- Enrich the catalog: 20-30 connectors covering agriculture and IoT
- First Z3 integration for verification of critical postconditions

Phase 3 — Distributed Execution (Months 6-9)

Objective: Reliable multi-node deployment with resilience.

- Implement the LLM-IR distribution protocol via WebSocket TLS
- Develop offline mode with local buffering and synchronization
- Implement parallel scheduler in SVM
- Complete fallback engine with 5 strategies
- Automated cross-compilation for embedded ARM Linux
- Observability dashboard (Grafana + Kafka audit events)

Phase 4 — Advanced LLM Calls (Months 9-12)

Objective: Multi-LLM orchestration and reasoning loops.

- Static LLM context construction system at compilation
- Multi-LLM pipeline with type validation between nodes

- Bounded reasoning loops with context enrichment
- Integration of local models (Llama, Mistral) for embedded Linux nodes
- Secret management via Vault with runtime injection

Phase 5 — Production and Certifications (Months 12-18)

Objective: Certifiable system, first production customer, start of certifications.

- Complete cryptographic audit trail with hash chain
- Support for Rust no_std microcontrollers (STM32, ESP32)
- Start MDR certification process for medical use
- Start IEC 61508 certification for critical industrial use
- First production customer deployment (agriculture or industry)
- Complete API documentation and catalog for integration partners

16. Technical Reference Resources

16.1 Fundamental Books

Compiler Theory

- "Crafting Interpreters" — Robert Nystrom (free online): practical implementation of a runtime
- "Engineering a Compiler" — Cooper & Torczon: IR and optimizations
- "Compilers: Principles, Techniques, and Tools" — Aho et al.: comprehensive theory
- "Types and Programming Languages" — Benjamin Pierce: formal type systems

Distributed Systems and Streaming

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

Rust and Systems

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

Virtual Machines

- "Virtual Machines" — Smith & Nair: instruction set design and runtimes

16.2 Essential Academic Papers

- "DSPy: Compiling Declarative Language Model Calls" — Khattab et al. (2023): closest to SCP
- "Neurosymbolic AI: The 3rd Wave" — Garcez & Lamb (2020): theoretical foundations
- "Constrained Language Models Yield Few-Shot Semantic Parsers" — Shin et al. (2021): constrained generation
- "Program Synthesis with Large Language Models" — Austin et al. (2021, Google Brain)
- "Tree of Thoughts" — Yao et al. (2023): tree-structured reasoning
- "ReAct: Synergizing Reasoning and Acting in Language Models" — Yao et al. (2022)

16.3 Key Libraries and Tools

Tool	Use in SCP
Outlines (Python)	Constraint generation LLM to compile to LLM-IR grammar
Z3 SMT Solver (Rust: z3 crate)	Formal verification of LLM-IR properties
Embassy (Rust)	Async runtime for Rust no_std microcontrollers
Tokio (Rust)	Async runtime for Rust Linux nodes
Protocol Buffers (prost crate)	Binary serialization of LLM-IR
LLVM IR Spec (reference)	Inspiration for the design of the LLM-IR format
probe-rs	Flashing and debugging Rust microcontrollers
BullMQ (NestJS)	Job queue for LLM-IR distribution