ClinCognition

# ClinClaw

## Workflow Manifest System

Governance & Operations for Hospital IT Administrators

| | |
|---|---|
| **Platform** | Microsoft Teams + Azure |
| **Target Audience** | Hospital IT & Clinical Informatics |
| **Status** | Production (v1.4.0) |
| **Date** | March 29, 2026 |
| **Prepared by** | Ernest Pedapati, MD |

## What Is a Workflow Manifest?

A workflow manifest is a single JSON file that declares everything about a clinical workflow: what triggers it, what inputs it needs, what it produces, how it's governed, and what messages the user sees at each stage. The manifest is the single source of truth -- C# handlers read from it and never hardcode workflow-specific strings. This means hospital administrators can understand, audit, and modify workflow behavior by reading JSON, not code.

### Why Manifests Matter in a Hospital Setting

Hospitals operate under regulatory constraints that most software never faces. A patient letter workflow touches protected health information, must produce output suitable for a clinician's signature, and generates artifacts that become part of the medical record. The people who understand these requirements -- clinical informaticists, compliance officers, division chiefs -- are not the same people who write C# code. Manifests bridge that gap. A compliance officer can read a manifest and verify that a workflow declares "sensitivityClass": "restricted" and "reviewRequirement": "human_required" without understanding the runtime implementation. An informaticist can adjust the LLM's generator instructions -- the reading level, the terminology preferences, the medical safety guardrails -- by editing JSON, not by filing a ticket with the engineering team and waiting for a release cycle.

This is not theoretical. ClinClaw currently runs 11 production workflows, each with a manifest that has been reviewed by the clinical team. When the neurology division wanted patient letters written at a 6th-to-8th-grade reading level with medication names explained in plain language, that change was a manifest edit -- not a code change, not a deployment, not a sprint. The generator instructions in patient-letter-draft.workflow.json were updated, the bot restarted, and every subsequent letter reflected the new guidance. That feedback loop -- clinical intent to production behavior in minutes, not weeks -- is what the manifest system exists to enable.

### Code-Driven vs Manifest-Driven

In a code-driven system, adding a workflow requires modifying C# files, recompiling, and redeploying. Every change carries the risk of introducing regressions in unrelated workflows. The people who request changes (clinicians, administrators) cannot verify what was actually changed without reading code diffs. Audit becomes "trust the developer" rather than "read the configuration."

In ClinClaw's manifest-driven system, a new workflow is a JSON file dropped into the Workflows directory. The catalog loads it at startup, the LLM discovers it as a callable tool, and the governance layer enforces its declared constraints -- no code changes required. The manifest is version-controlled in git alongside the code, so every change has a commit message, a timestamp, and an author. When a compliance audit asks "who changed the patient letter reading level and when?", the answer is a git log entry, not a memory.

### Manifest Sections (20 Total)

| Section | Required | Purpose |
|---|---|---|
| id / version / displayName | Yes | Identity: unique ID, semantic version, human-readable name |
| description / owningTeam / category | Yes | Metadata: what the workflow does, who owns it, classification |
| trigger | Yes | Intents, utterance patterns, conversation scopes, explicit invocation flag |
| prerequisites | Yes | Required services, session capabilities, required inputs |
| input | Yes | Field definitions: name, type, required flag, default value |
| output | Yes | Output fields, OneDrive folder template, filename template |
| execution | Yes | Mode (executor/in_process), job type, capabilities, timeout |
| metadataDefaults | Yes | Workflow type, owner team, service line, data domain |
| userExperience | Yes | Submission messages, resume templates, result messages |
| governance | Yes | Required metadata, review requirement, promotion policy, audit events |
| authoring | Optional | LLM engine, interpreter instructions, generator instructions |
| rendering | Optional | Rendering engine, output format, model, size |
| preflight | Optional | Confirmation card: fields, toggles, dropdowns, bindings |
| surfaces | Optional | Admin portal panel contributions (meeting, archive, actions) |
| entitlements | Optional | RBAC: workflow key, screen keys for Entra group gating |
| outreach | Optional | SMS template, clinic name, contact limits, response deadlines |

## The Universal Template

The universal template (docs/universal-workflow-template.workflow.json) documents all 20 sections with inline _note annotations explaining every field, valid enum values, and validation rules. Copy it, delete sections you don't need, fill in your values. The hello_world manifest demonstrates the minimum viable workflow: one input field, one output field, in-process execution, 10-second timeout, and canonical governance values.

### Example: Minimal Workflow (hello_world)

The simplest valid manifest. One input, one output, in-process execution, 10-second timeout. Uses canonical governance values (schemaVersion 1.1, reviewRequirement "none", sensitivityClass "internal"). This is the starting point for any new workflow.

| Field | Value |
|---|---|
| id | hello_world |
| schemaVersion | 1.1 |
| execution.mode | in_process |
| execution.timeoutSeconds | 10 |
| input.fields | greeting (string, optional, default: "Hello, World!") |
| output.fields | echo_response (string, required) |
| governance.reviewRequirement | none |
| governance.sensitivityClass | internal |
| trigger.conversationScopes | personal, directline |
| trigger.requiresExplicitInvocation | true |

### Example: Full-Featured Workflow (patient_letter_draft)

The canonical reference manifest. Uses all major sections: LLM interpreter (parses user request into structured fields), LLM generator (ghost-writes the letter with medical safety rules), preflight card (confirmation with toggles for medication list and diagnosis history), file output templates (OneDrive folder by patient MRN), and executor-based async execution.

| Section | Key Configuration |
|---|---|
| authoring.interpreter | Extracts patient_mrn, letter_request, tone from untrusted user input. 300 tokens, 30s timeout. Regex fallback patterns for MRN and patient name. |
| authoring.generator | Ghost-writes at 6th–8th grade reading level. 1,200 tokens, 45s timeout. JSON schema: subject, greeting, body_paragraphs, closing, signature, assumptions. |
| preflight | 6 fields: patient MRN (text), patient name (text), letter purpose (textarea), include medications (toggle, binding injects "Include medications by name, dose, frequency"), include diagnoses (toggle), additional notes (textarea with prefix binding). |
| output | folderTemplate: ClinClaw/Patients/{patient_mrn}-{patient_name}/{year}-{month}. filenameTemplate: {patient_first_initial}{patient_last_name}_{letter_purpose}_{timestamp}.docx |
| execution | Mode: executor. Job type: docx_review. Timeout: 180s. Capabilities: docx_review, bounded_subprocess, artifact_read, artifact_write, ai_content_generation. |
| governance | reviewRequirement: human_required. sensitivityClass: restricted. Audit events: workflow_requested, workflow_queued, workflow_completed. |

### Example: Command Manifest (epic.command.json)

System commands use a simpler manifest shape -- no authoring, no execution pipeline. The Epic connect command is the only "restricted" command because it initiates access to patient data.

| Field | Value |
|---|---|
| id | epic_connect |
| slashName | epic |
| handler | epic_auth_connect |
| category | auth |
| teamsCommandList | true (appears in / autocomplete) |
| priority | 10 (high -- near top of autocomplete) |
| governance.sensitivityClass | restricted |
| governance.auditEvent | epic_auth_initiated |
| governance.entitlementKey | command.epic_connect |
| governance.requiredServices | Epic SMART on FHIR |

### Schema Versioning

Every manifest declares a schemaVersion field. Supported versions: "1.0" (legacy, applied automatically to manifests without the field) and "1.1" (current, with enum validation and canonical governance values). The catalog rejects unknown versions at startup. When the schema evolves, bump the version and update validation rules -- old manifests continue to load via the backward compatibility layer (ManifestBackwardCompatibility.cs), which is a single deletable file with one call site.

### Governance Enum Values

| Field | Valid Values | Meaning |
|---|---|---|
| reviewRequirement | human_required, read_only_summary, none | Whether output needs human review before delivery |
| promotionPolicy | not_promoted, not_promoted_until_reviewed, manual | Whether output can be promoted to institutional use |
| sensitivityClass | restricted, confidential, internal | Data sensitivity: PHI, internal sensitive, non-sensitive |

## The Message Chain Architecture

Every Teams message passes through a 3-phase pipeline: classify, resolve, execute. Phase 1 is deterministic and costs nothing. Phase 2 uses the LLM only when needed. Phase 3 enforces manifest governance before and after handler execution.

### Why This Architecture Exists

The previous architecture routed messages through 760 lines of hardcoded keyword patterns -- 25 sequential string-matching functions checked in order, with known collision problems. Adding a workflow meant modifying four C# files and redeploying. The LLM was bolted on as an optional "refinement layer" that could only agree or disagree with the keyword router's decision. System commands (auth flows, template management) and clinical workflows (patient letters, chart summaries) were mixed in a single 134-line if-chain with no distinction in trust model or governance.

The new architecture separates concerns that should never have been coupled. System commands are plumbing -- they don't touch patient data, don't need audit trails, and should dispatch instantly. Workflows are clinical operations -- they touch PHI, need governance, and benefit from LLM understanding of natural language. Knowledge questions are information

retrieval -- they go straight to the search engine. Mixing all three in one routing path meant every message paid the cost of the most complex path, and governance checks applied unevenly.

The 3-phase pipeline makes these boundaries explicit. Phase 1 is a cheap triage -- under 1ms, no LLM call, just structural analysis. Phase 2 uses the LLM only for the messages that need semantic understanding (workflow requests), while system commands and protocol messages skip it entirely. Phase 3 applies governance uniformly to every workflow invocation, regardless of how it was routed. The result: system commands are faster (no LLM overhead), workflows are smarter (LLM picks the right one and extracts parameters), and governance is consistent (every workflow goes through the same gate).

### Phase 1: Message Classification

| Class | Signal | Dispatch Path |
|---|---|---|
| Protocol | Attachments, empty text, slash prefix | Direct handler (file upload, ignore) |
| SystemCommand | Registered slash command or intent | SystemCommandDispatcher (auth, templates, debug) |
| WorkflowRequest | Action language or tool-call result | WorkflowDispatcher → GovernanceGate → Handler |
| KnowledgeQuestion | Everything else | RAGFlow semantic search or agent orchestrator |

### Phase 2: Intent Resolution

For workflow requests and knowledge questions, the LLM tool-calling router (ToolCallingIntentRouter) receives the message and a tool catalog auto-generated from loaded manifests. Each manifest becomes a callable tool named invoke_{workflow_id} with parameters from the manifest's input fields. The LLM picks the right workflow and extracts structured parameters in a single call. If the LLM fails, the system falls back to manifest-driven utterance pattern matching -- also manifest-driven, no hardcoded patterns needed.

### Phase 3: Governed Execution

Before the handler runs, the GovernanceGate evaluates manifest constraints (scope, capabilities, explicit invocation). The handler runs within a CancellationToken bounded by execution.timeoutSeconds. After completion, the WorkflowAuditEventEmitter verifies that all governance.requiredAuditEvents were emitted as real database records. If governance.reviewRequirement is "human_required", the ReviewGate sends a review notice to the provider.

### Old Chain vs New Chain

| Metric | Before | After |
|---|---|---|
| Routing code | 760 lines (hardcoded keywords) | 130 lines (classifier + registry) |
| Files to change for new workflow | 4+ C# files + redeploy | 1 JSON file + restart |
| Governance fields enforced | 4 of 12 | 10 of 12 |
| Audit events | Structured log lines | Database records with verification |
| Review enforcement | Display only | Output held until reviewed |
| System command dispatch | 134-line if-chain | Dictionary with 13 entries |

## Slash Commands & Command Manifests

System commands (auth, templates, simulation, debug) are dispatched via slash commands -- deterministic, no LLM needed. Each command has a manifest (*.command.json) declaring its slash name, handler, governance fields, and audit event. The SystemCommandDispatcher reads from the command manifest catalog at startup. Adding a command: write the handler, drop a JSON file, restart.

| Command | Category | Description | Audit Event |
|---|---|---|---|
| /help | system | Show available commands | -- |
| /epic | auth | Connect to Epic (SMART on FHIR) | epic_auth_initiated |
| /m365 | auth | Connect to Microsoft 365 | m365_auth_initiated |
| /m365-disconnect | auth | Disconnect Microsoft 365 | m365_auth_revoked |
| /calendar | productivity | Check schedule availability | calendar_checked |
| /event | productivity | Create a calendar event | calendar_event_created |
| /email | productivity | Check or summarize inbox | email_checked |
| /templates | templates | List saved stationery | -- |
| /save-template | templates | Save current template | -- |
| /uploads | files | Last upload status | -- |
| /source | files | Download source file | -- |
| /simulate | debug | Simulate provider identity | simulation_started |
| /debug | debug | Retrieval scope debug info | -- |

### Command Manifest Governance

Each command manifest declares a governance block with sensitivityClass ("internal" or "restricted"), an optional auditEvent name emitted as a real database record on execution, an optional entitlementKey for RBAC via Entra groups, and requiredServices that gate availability. The Epic connect command is the only "restricted" command with an entitlement key -- access can be revoked from the admin panel without code changes.

## Governance Enforcement

The governance layer enforces manifest-declared constraints at three points: before execution (GovernanceGate), during execution (timeout enforcement), and after execution (audit verification and review gate). Every governance field is either actively enforced or explicitly deferred with a tracking issue.

### The Problem Governance Solves

When an AI system generates a clinical document, three questions must have clear answers: who authorized this workflow to run, what constraints governed the output, and where is the audit trail? In many AI deployments, the answers are "the developer hardcoded it," "whatever the model decided," and "the application log, if someone thought to check." That is not acceptable in a hospital where generated documents may be signed by physicians, sent to patients, or filed in medical records.

ClinClaw's governance layer makes these answers concrete and verifiable. Authorization comes from the manifest's entitlement keys mapped to Entra ID groups -- the admin panel shows exactly which providers can run which workflows. Constraints come from the manifest's governance section -- sensitivity classification, review requirements, and promotion policies are declared in JSON that compliance officers can read. The audit trail consists of named events written to the audit database with correlation IDs that link every step of a workflow invocation. When a regulator asks "show me the controls around AI-generated patient letters," the answer is the manifest file, the audit query, and the admin panel -- not a code review.

### Honest About What's Deferred

Not every governance field is enforced today. The table below is transparent about which fields are active enforcement points and which are deferred. Deferred fields are validated at startup (invalid values fail deployment) and visible in the admin panel, but don't block execution yet. Each deferred field has a tracking issue with a concrete implementation plan. We chose to ship the governance schema complete and enforce incrementally rather than wait for full enforcement before deploying any governance. The schema is the contract; enforcement catches up.

| Manifest Field | Status | Enforcement Mechanism |
|---|---|---|
| execution.timeoutSeconds | Enforced | Linked CancellationToken cancels handler on timeout |
| governance.sensitivityClass | Enforced | Propagated to executor job metadata and audit records |
| governance.requiredAuditEvents | Enforced | Database records emitted, post-execution verification |
| governance.reviewRequirement | Enforced | ReviewGate sends review notice for human_required |
| trigger.conversationScopes | Enforced | ReadinessEvaluator blocks wrong conversation types |
| prerequisites.* | Enforced | ReadinessEvaluator checks services, capabilities, inputs |
| trigger.requiresExplicitInvocation | Deferred | Blocked until keyword router removed (tracked) |
| execution.allowedCapabilities | Deferred | Validated at startup, runtime check pending |
| governance.promotionPolicy | Deferred | Displayed in admin panel, no enforcement logic yet |
| governance.requiredMetadata | Deferred | Validated for non-emptiness, not checked at runtime |

## Provider Context Layers

Every workflow receives a merged context from four layers, each with different ownership and override semantics. Institution and division rules are mandatory -- providers can't override them. Specialty and user preferences are provider-controlled and customizable.

| Layer | Owner | Content | Storage |
|---|---|---|---|
| Institution | System admin | Compliance rules, reading level, legal disclaimers, terminology | PostgreSQL (singleton, versioned) |
| Division | Division lead | Discipline rules, workflow mandates, required content | PostgreSQL (per division ID, versioned) |
| Specialty | Provider | Clinical discipline context, terminology guidance, personal notes | PostgreSQL (per user + specialty) |
| User | Provider | Display name, credentials, tone, signature, EMR instructions | PostgreSQL + Graph/Epic profile |

### Template Management

Providers manage templates (letterhead, slide themes, report templates, clinical note templates) in a "My Templates" card in the Teams personal tab. Each template is a tile with a thumbnail, name, type badge, and star icon for the active version. Upload is drag-and-drop. Previous versions are preserved for rollback. Templates are stored in MinIO with metadata in PostgreSQL -- the same infrastructure used for workflow artifacts.

## Inline LLM Guard (Planned)

ClinClaw's message chain currently enforces governance at the manifest level: who can run a workflow, what timeout applies, whether human review is required. What it does not yet do is inspect the content of messages and LLM responses

at the token level. A provider could paste a prompt injection payload into a letter request. The LLM could hallucinate a patient identifier that looks real. A generated letter could contain language that violates institutional tone guidelines. These are content-level risks that manifest governance cannot address.

### LLM Guard for .NET

llmguard-dotnet is a pure C#/.NET 8 port of Protect AI's open-source llm-guard library, developed in-house with 133 test methods across 15 test files. The core contract is simple: every scanner implements IInputScanner.Scan(string prompt) and returns a ScanResult with the sanitized text, a pass/fail boolean, a risk score (0.0 to 1.0), and a list of identified risks. Scanners compose into a sequential pipeline where each scanner's sanitized output feeds the next. ML-backed scanners (PromptInjection, Toxicity, BanCode, BanTopics, Gibberish) run native ONNX inference via Microsoft.ML.OnnxRuntime with no Python dependency.

### Implemented Scanners

| Scanner | Mechanism | Healthcare Relevance |
|---|---|---|
| Anonymize | Regex recognizers + Vault for placeholder round-trip. Supports en/zh. Redacts to [REDACTED_TYPE_N] with stable per-entity indices. | Strips MRN, SSN, phone, email, names before LLM sees them. Vault restores real values in output. |
| PromptInjection | DeBERTa-v3 ONNX model. Match types: FULL, SENTENCE, CHUNKS, TRUNCATE_HEAD_TAIL. Threshold 0.92 default. | Catches jailbreak attempts in clinical prompts. Sentence mode handles long narratives where injection is buried mid-text. |
| Toxicity | RoBERTa ONNX multi-label classifier. 7 labels: toxicity, severe, obscene, threat, insult, identity_attack, sexual. FULL/SENTENCE modes. | Flags inappropriate language in patient-facing letters. Catches tone violations before delivery. |
| Secrets | Native regex detectors for AWS, GitHub, JWT, OpenAI, Slack, Stripe, Azure keys. Redact modes: ALL, PARTIAL, HASH. | Prevents credential leakage if a provider accidentally pastes API keys or tokens into a message. |
| BanSubstrings | String matching with case-insensitive and word-boundary options. | Block institutional terms that should not appear in patient correspondence. |
| Regex | Configurable patterns. is_blocked=true (forbidden) or false (required). Optional redaction. | Enforce MRN format rules, require disclaimer text, redact date-of-birth patterns. |
| InvisibleText | Unicode zero-width character detection. | Catches hidden prompt injection via invisible Unicode characters in pasted clinical text. |
| TokenLimit | Tokenizer-based max token enforcement. | Prevents oversized prompts that could degrade LLM quality or exceed context windows. |
| BanCode | ONNX code detection model. | Blocks code submission in clinical workflows where code is never expected. |
| BanTopics | Zero-shot ONNX classifier against topic blacklist. | Block treatment recommendations or drug dosing in administrative-only workflows. |
| Gibberish | ONNX gibberish detection model. | Catches corrupted or nonsensical input that would waste LLM resources. |
| Code | ONNX code fragment detection. | Flags code in contexts where it might indicate a technical attack vector. |

### Integration Into the Message Chain

The guard pipeline inserts at two points in the existing chain. On the input side, it runs after Phase 2 (intent resolution) and before Phase 3 (governed execution). The user's message and any extracted parameters pass through the input scanners before the handler sees them. On the output side, it runs after the handler produces content and before the result is delivered to the user or written to a DOCX artifact. This means a prompt injection hidden in a patient letter request is caught before the generator LLM processes it, and PII hallucinated by the generator is redacted before the letter reaches the provider.

The scanner list is workflow-specific and manifest-driven. A planned governance.scanners section in the workflow manifest would declare which scanners to apply and their thresholds. A patient letter workflow might enable Anonymize, PromptInjection, Toxicity, and FactualConsistency. A leadership meeting summary might enable only BanTopics (to prevent leaking HR-sensitive content). An administrative SOP draft might skip PII scanners entirely since it contains no patient data.

The manifest controls the scanner list, keeping handlers generic.

### Implementation Status

The llmguard-dotnet library has 12 input scanners implemented with 133 test methods. The core scanners (PromptInjection, Toxicity, Anonymize, Secrets) are complete with ONNX inference and upstream test parity. Remaining work: output scanners (Deanonymize, FactualConsistency, Sensitive), the pipeline orchestrator (scan_prompt/scan_output equivalents), and the NuGet package for ClinClaw integration.

Integration into ClinClaw will be a two-step process: register the scanner pipeline as a service in DI, and add input/output guard calls to InvokeWithTimeoutAsync in the WorkflowDispatcher. The existing GovernanceGate, AuditEventEmitter, and ReviewGate remain unchanged. The guard adds content-level protection (what the LLM sees and produces) to the governance-level protection (who can run what, with what constraints) already in place. Together they cover the full risk surface: manifest governance prevents unauthorized execution, LLM Guard prevents unsafe content.

## Deployment Phases

ClinClaw deploys in three phases, each independently valuable. An administrator can stop at any phase and have a working, useful system. Every capability can be disabled independently.

### Why Phased Deployment Matters

Hospital IT teams are rightly cautious about AI systems that touch clinical data. A big-bang deployment that goes from zero to "AI writes notes in Epic" in one step is a non-starter for most institutions. Phased deployment lets the IT team build confidence incrementally: Phase 1 proves the platform works with zero clinical risk (document Q&A; only, no patient data). Phase 2 adds read-only Epic integration for document generation -- the AI reads charts but never writes back. Phase 3 adds write-back capabilities (SMS outreach, note writing) only after the institution has validated the platform's governance, audit, and access controls in production.

Each phase has its own exit strategy. Phase 1 can be decommissioned by removing the Teams app -- no residual data in any clinical system. Phase 2 adds read-only Epic scopes that can be revoked from the Epic admin console without touching ClinClaw. Phase 3 capabilities are each gated independently: disabling outreach doesn't affect letter drafting, disconnecting Epic doesn't break document Q&A.; The manifest system makes this granularity possible -- each workflow declares its own prerequisites, and the readiness evaluator blocks execution when prerequisites aren't met rather than failing the entire system.

| Phase | Timeline | Risk | Capabilities | Requirements |
|---|---|---|---|---|
| 1. Knowledge Base | 2–3 weeks | Low | Document Q&A; with cited answers | Azure, Entra ID, Teams app, VM, PostgreSQL |
| 2. Document Generation | 2–4 weeks | Mediu m | Patient letters, presentations, meeting summaries | Epic SMART on FHIR (read-only), executor service |
| 3. Clinical Write-Back | 4–8 weeks | High | SMS outreach, scheduling, Epic note writing | ACS, Epic write scopes, CLI certification |

### Administrator Controls

At every phase, the administrator controls: access (Entra groups for users, workflows, and admin roles), governance (sensitivity class, review requirements, audit events -- all in manifests, all visible in the admin panel), data (OneDrive folder structure, audit retention, artifact metadata), and risk (every workflow and EMR integration disabled independently, graceful degradation -- disabling Epic doesn't break document Q&A;).