

NOVA AI

Technical Proposal

Personal AI, On Personal Devices.

Version: 1.2.4
License: Apache 2.0
Repository: github.com/Hamza35779/NOVA-AI
Documentation: hamza35779.github.io/NOVA-AI
Python: 3.10 - 3.13
Date: September 2026

1. Abstract

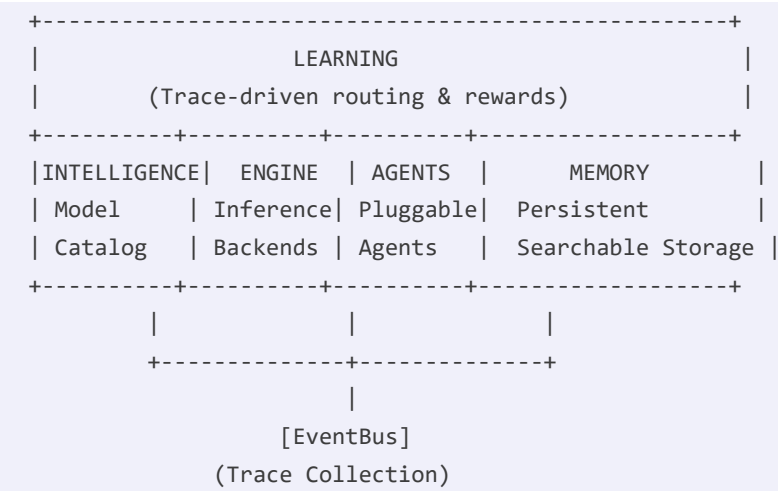
NOVA AI is a local-first personal AI agent framework built in Python 3.10+ with Rust (PyO3) performance extensions and a TypeScript/Tauri desktop frontend. It provides a modular, extensible stack organized around five core primitives — Intelligence, Engine, Agentic Logic, Memory, and Learning — connected through trace-driven feedback loops.

The framework ships with 8+ agent types, 58+ built-in tools, 5 memory backends, multiple inference engine backends (Ollama, vLLM, SGLang, llama.cpp, Cloud), full MCP (Model Context Protocol) integration, and a comprehensive CLI with a web-based workstation UI. It is designed to run AI agents locally by default, calling cloud APIs only when query complexity warrants it, with per-query cost tracking and automatic fallback.

2. System Architecture

2.1 Five-Primitive Architecture

The system is organized around five core abstractions that communicate through a thread-safe pub/sub EventBus defined in core/events.py. The primitives form a directed dependency graph that creates a feedback loop: agents produce traces, traces inform learning, learning improves routing, and better routing improves agent performance.



2.2 Registry Pattern

All extensible components use a decorator-based registry for runtime discovery. Registration happens at import time — no factory function or configuration file needs modification.

```
@EngineRegistry.register("ollama")
class OllamaEngine(InferenceEngine):
    ...
```

Typed registries exist for:

Registry	Type Parameter	Purpose
ModelRegistry	Any (ModelSpec)	Model metadata
EngineRegistry	Type[InferenceEngine]	Inference backends
MemoryRegistry	Type[MemoryBackend]	Memory backends

AgentRegistry	Type[BaseAgent]	Agent implementations
ToolRegistry	Any (BaseTool classes)	Tool implementations
ChannelRegistry	Any (BaseChannel classes)	Channel implementations

2.3 Source Layout

src/nova_ai/	
core/	Registry, types, config, events
intelligence/	Model catalog, router
engine/	Inference backends (Ollama, vLLM, Cloud, ...)
agents/	Agent implementations (8+ types)
tools/	Tool implementations (58+ tools)
memory/	Memory backends (SQLite, FAISS, ColBERT, ...)
learning/	Router policies, rewards, SkillForge
traces/	Trace store, collector, analyzer
telemetry/	Inference metrics, aggregation
server/	FastAPI HTTP server (OpenAI-compatible)
mcp/	MCP client/server/adapters
sandbox/	Docker/Podman agent sandbox
channels/	Messaging channel integrations
scheduler/	Cron/interval task scheduler
cli/	Click-based CLI commands
security/	Guardrails, scanning, audit

3. Inference Engine Layer

3.1 Engine Interface

All backends implement the InferenceEngine ABC with a uniform interface:

```
class InferenceEngine(ABC):
    def generate(self, messages, *, model, **kwargs) -> dict: ...
    def stream(self, messages, *, model, **kwargs) -> Iterator[str]: ...
    def list_models(self) -> list[str]: ...
    def health(self) -> bool: ...
```

3.2 Backend Implementations

Backend	Type	Technology	Use Case
Ollama	Local	Native HTTP API	Default local engine, broad model support
vLLM	Local	OpenAI-compatible API	High-throughput serving with PagedAttention
SGLang	Local	OpenAI-compatible API	Structured generation with constrained decoding
llama.cpp	Local	OpenAI-compatible API	GGUF quantized models, CPU/GPU hybrid
MLX	Local	OpenAI-compatible API	Apple Silicon optimized inference
Cloud	Cloud	Provider SDKs	OpenAI, Anthropic, Google Gemini, DeepSeek

3.3 Smart Model Router

The MultiEngine accepts model='auto': the last user message is scored by score_complexity(), simple queries stay on the fastest local engine, and queries at or above the auto_route_threshold (default 0.55) escalate to the first available cloud model from cloud_model_preference (claude -> gpt -> gemini -> deepseek). When cloud is unreachable, complex queries fall back to local instead of failing.

Per-query cost tracking: OpenRouter's reported per-request cost when present, otherwise an estimate from the pricing table. Local inference always costs \$0.00. Cost flows through InstrumentedEngine into telemetry records and persisted Trace.total_cost_usd.

3.4 Engine Discovery

discover_engines() probes all registered backends for health at startup, returning healthy engines sorted with the user's configured default first. The system automatically falls back to any available engine if the preferred one is unavailable.

4. Agent Framework

4.1 Class Hierarchy

```
BaseAgent (ABC)
+-- SimpleAgent      (single-turn, no tools)
+-- OpenHandsAgent   (wraps openhands-sdk)
+-- ClaudeCodeAgent  (Claude Agent SDK via Node.js)
+-- SandboxedAgent   (wraps any BaseAgent in Docker)
+-- ToolUsingAgent    (accepts tools, ToolExecutor)
    +-- OrchestratorAgent (multi-turn tool-calling loop)
    +-- NativeReActAgent (Thought-Action-Observation)
    +-- NativeOpenHandsAgent (CodeAct code execution)
    +-- RLMAgent       (recursive LM with REPL)
    +-- OperativeAgent  (persistent scheduled agent)
    +-- MonitorOperative (long-horizon monitoring)
```

4.2 BaseAgent Contract

```
class BaseAgent(ABC):
    agent_id: str
    accepts_tools: bool = False

    def run(self, input: str, context: Optional[AgentContext] = None,
            **kwargs) -> AgentResult: ...
```

AgentContext provides: conversation history, tool names, memory results, metadata.

AgentResult returns: content, tool_results, turns, metadata.

4.3 OrchestratorAgent (Default)

The OrchestratorAgent is the primary/default agent served via nova serve. It implements a tool-calling loop in two modes:

- function_calling (default): Uses OpenAI-format tool definitions. Parses tool_calls from engine response. Supports parallel tool execution via ThreadPoolExecutor.
- structured: Uses THOUGHT/TOOL/INPUT/FINAL_ANSWER text protocol. This is the format used by SFT/GRPO training pipelines, making the Orchestrator a trainable agent type.

Loop guard: degenerate-loop detection via SHA-256 hash tracking, ping-pong detection, per-tool budgets, context overflow recovery.

4.4 Tool-Calling Loop

1. Build messages + tool definitions (OpenAI format)
2. Call engine.generate() with messages
3. If response has tool_calls -> execute each tool via ToolExecutor
4. Append tool results as TOOL messages

5. If no tool_calls -> return final answer
6. If max_turns exceeded -> return with warning
7. Goto step 2

5. Tool System

5.1 BaseTool Interface

```
class BaseTool(ABC):
    tool_id: str

    @property
    def spec(self) -> ToolSpec: ...

    def execute(self, **params) -> ToolResult: ...

    def to_openai_function(self) -> Dict[str, Any]: ...
```

5.2 Tool Categories (58+ Tools)

Web:	web_search, browser_navigate, browser_click, browser_type, browser_extract, browser_screenshot, http_request, web_readability
Code:	code_interpreter, code_interpreter_docker, shell_exec, repl, code_scaffolder, docker_shell_exec
Files:	file_read, file_write, apply_patch, git_tool, git_manager, file_converter
Knowledge:	knowledge_search, knowledge_sql, scan_chunks, retrieval, memory_manage, memory_wiki_tools, storage_tools
Media:	image_tool, audio_tool, text_to_speech, screen_capture, screen_monitor
System:	system_monitor, scheduler_tool, canvas_tool, clipboard_ai, api_tester
Data:	data_analyzer, db_query, pdf_tool, doc_generator
Integration:	mcp_adapter, cisco_packet_tracer, skill_manage, user_profile_manage, approval_store, proactive_tools, channel_tools, digest_collect

5.3 MCP Integration

Full Model Context Protocol (MCP) client/server implementation:

- MCPClient connects to external MCP servers via transports
- MCPToolAdapter wraps external tools as native BaseTool instances
- MCPToolProvider dynamically discovers and registers MCP-hosted tools
- Protocol version: 2025-03-26
- MCP-discovered tools are seamlessly integrated into agents through ToolExecutor

5.4 ToolExecutor

The ToolExecutor handles tool dispatch with:

- JSON argument parsing from model output
- Latency tracking per tool call
- EventBus integration (TOOL_CALL_START / TOOL_CALL_END events)
- OpenAI function-calling format conversion via get_openai_tools()

6. Memory & Retrieval

6.1 Memory Backend Interface

```
class MemoryBackend(ABC):
    def store(self, document, metadata) -> str: ...
    def retrieve(self, query, k=5) -> list[RetrievalResult]: ...
    def delete(self, doc_id) -> bool: ...
    def count(self) -> int: ...
```

6.2 Backend Implementations

Backend	Description
SQLite/FTS5	Zero-dependency default. Full-text search with FTS5. Good for quick start.
FAISS	Dense vector retrieval via Facebook AI Similarity Search. Requires sentence-transformers.
ColBERTv2	Late interaction retrieval. Higher precision than FAISS for complex queries.
BM25	Classic Okapi BM25 term-frequency retrieval. Fast, no embeddings needed.
Hybrid	Reciprocal Rank Fusion combining sparse (BM25) + dense (FAISS) results.

6.3 Ingestion Pipeline

Document ingestion follows a pipeline: file reading -> chunking (ChunkConfig) -> embedding generation (SentenceTransformerEmbedder) -> storage in the chosen backend. When `agent.context_from_memory` is enabled, relevant documents are retrieved and prepended to the prompt with source attribution.

6.4 Context Injection

The `inject_context()` function retrieves relevant chunks from the memory backend and prepends them to the agent's prompt. Each chunk includes source attribution (file path, chunk index) so the agent can cite its sources.

7. Learning & Trace System

7.1 Trace Schema

Every agent interaction produces a Trace capturing the full sequence of steps — routing decisions, memory retrieval, inference calls, tool invocations, and final responses. Traces are persisted in SQLite via TraceStore.

7.2 Trace-Driven Learning

The TraceAnalyzer computes statistics from accumulated traces (latency, cost, tool usage, success rates). The TraceDrivenPolicy uses these statistics to learn which model/agent/tool combinations produce the best outcomes for different query types.

7.3 Router Policies

Policy	Description
HeuristicRouter	Default. Scores query complexity, routes simple to local, complex to cloud.
TraceDrivenPolicy	Learns from trace outcomes. Uses aggregated stats for routing decisions.
GRPORouterPolicy	Group Relative Policy Optimization. Future RL-based routing.

7.4 SkillForge

SkillForge is the skill synthesis pipeline that auto-creates skills from repeated tool patterns:

- Miner: discovers repeated tool-calling patterns from traces
- Synthesizer: turns mined patterns into reusable skill manifests (TOML) via local LLM
- Pipeline: end-to-end skill creation
- Gauntlet: validation of synthesized skills
- Store: persistent skill storage
- Adoption: tracks skill usage and effectiveness

7.5 Training Pipelines

NOVA AI includes training pipelines for improving the Orchestrator agent:

- SFT (Supervised Fine-Tuning): trains on trace data
- GRPO (Group Relative Policy Optimization): RL-based training
- DPO (Direct Preference Optimization): preference-based training
- LoRA (Low-Rank Adaptation): efficient fine-tuning
- Data pipeline: trace -> training data preparation
- Deploy: trained model deployment

8. Security Model

8.1 Loop Guard

The LoopGuard detects degenerate tool-calling loops via:

- SHA-256 hash tracking of consecutive states
- Ping-pong detection (A -> B -> A -> B ...)
- Per-tool call budgets
- Context overflow recovery (4-stage compression)

Both Python and Rust backends available.

8.2 Capability Policy / RBAC

Tools declare `required_capabilities`. Agents need matching permissions. The `OperatorManager` checks capability requirements before activation. Capability names are validated against the `Capability` enum so manifest typos surface as warnings.

8.3 Security Scanning

- `SecretScanner`: detects hardcoded API keys and secrets
- `PIIScanner`: detects personally identifiable information
- `Boundary Guard`: scans external tool arguments for security threats
- `SSRF Protection`: web tools check for server-side request forgery
- `Taint Checking`: tracks data provenance through tool pipelines
- `Redaction-before-cloud`: scrubs secrets/PII before cloud transmission

8.4 Budget Enforcement

Per-agent cost/token limits with automatic pause on exceed. Tracked via `TelemetryStore`. Per-operator rate limiting (`rate_limit_rpm` field). Failure circuit breaker with `max_consecutive_failures`.

9. Deployment

9.1 Windows Installer

Self-contained Inno Setup installer built from PyInstaller onedir bundle:

- Per-user install to %LOCALAPPDATA%\Programs\NOVA AI
- No uv, git, or Python required for end users
- Start Menu + optional desktop shortcuts
- Optional PATH integration
- Proper uninstaller (preserves user data in ~/.nova_ai)

9.2 Docker

Containerized deployment with docker-compose:

- Dockerfile: multi-stage build with Python + Node.js
- Dockerfile.gpu: NVIDIA GPU support
- Dockerfile.sandbox: sandboxed agent execution
- docker-compose.yml: full stack with Ollama sidecar

9.3 Desktop Application

Tauri 2.x desktop app:

- Native window with real-time status
- Ollama integration for local model management
- Model Hub with curated catalog and background installation
- Auto-update via GitHub releases
- Built-in chat UI with voice support

9.4 Channel Integrations

Built-in channels (15+): Telegram, Discord, Slack, WhatsApp, Line, Viber, Messenger, Reddit, Mastodon, XMPP, Rocket.Chat, Zulip, Twitter/X, Twitch, Nostr, Twilio, Gmail. Each channel implements the BaseChannel ABC and registers via @ChannelRegistry.register().

10. Performance Benchmarks

10.1 Benchmark Framework

The bench/ module provides a pluggable benchmarking framework:

- LatencyBenchmark: per-call latency measurement
- ThroughputBenchmark: tokens/second measurement
- BenchmarkSuite: configurable test suites
- All benchmarks register via @BenchmarkRegistry.register()

10.2 Cost Tracking

Every cloud inference call tracks:

- Per-query USD cost (from OpenRouter or pricing table estimate)

- Token counts (input/output)
- Latency (time to first token, total)
- Local inference always costs \$0.00

Cost data flows through InstrumentedEngine into telemetry and trace stores.

10.3 Energy Monitoring

Energy monitoring via pynvml (NVIDIA), amdsmi (AMD), zeus-ml (Apple Silicon). Tracks GPU power consumption during inference for energy-aware routing.