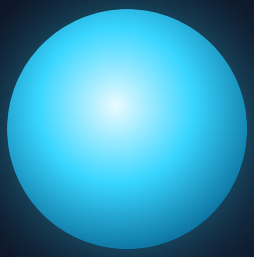

JARVIS OS

System Architecture — Full Reference

- Personal AI operating layer for Windows — voice + HUD, multi-brain LLM
- OODA command pipeline · Class A/B permission governance
- Self-upgrade pipeline (research → Claude → security review → confirm)
- Persistent cross-session memory · autonomy (Iron Man) · browser + system control

Generated architecture reference · Python · single-user local agent

1. Layered Overview

Five bands: channels take input, the orchestrator runs OODA, cognition/routing decide, governance gates every risky act, tools & agents execute.

```
CHANNELS (entry points)
  main.py -> web_server.py (HUD :8765) . terminal_channel.py
             remote_server.py (LAN, token) . gui.py . voice_input/output
                                |
                                v
ORCHESTRATOR  (core/orchestrator.py) - the OODA brain
  handle() -> single-flight -> classify -> _dispatch -> act -> remember
   |             |              |              |
   v             v              v              v
 REASONING     AGENTIC        LLM LAYER     PERMISSION / SAFETY
 reasoning_    agentic_core   llm_client    permission_gate
  core (rules) (JSON+ReAct)   (multi-brain) safety_guard . permission_manager
   |             |              |              |
   v             v              v              v
 ROUTING/VERIFY MEMORY         AUTONOMY       SELF-EXTENSION
 query_router   memory_manager autonomy_      self_upgrade
 verification_  conversation_    daemon        self_improve
  council        memory        mission         _self_code
 council        experience     proactive_advisor
                               |
                               v
        TOOLS (tool_registry) ---- AGENTS (agent_registry)
        browser_root, windows_app,   FRIDAY, TRON, ULTRON,
        screen/vol/power/network/    IRA, MYTHOS + business
        input/process, trading,      agents
        news, web, utility, youtube,
        email, file_ops, vision
```

2. Request Lifecycle (OODA)

```
user text
  |
  v  handle()   [SINGLE-FLIGHT: _handle_lock + _turn_epoch -> only newest turn answers]
  |
  +- Observe/Orient:  reasoning_core.classify(text) -> Intent{name, agent, args}
  |                   (rule table, first-match wins, wake-word stripped)
  |
  v  _dispatch(intent)
  |
  +- direct handler   -> tool call (open_app, volume_up, news, settings, ...)
  |
  +- planner action   -> _run_action() -> permission_gate.classify()
  |                        CLASS_A run . CLASS_B hold . FORBIDDEN refuse
  |
  +- unknown/freeform -> _freeform():
        |
        +- query_router.route() -> DIRECT | LIVE | DEEP
        |     DEEP -> verification_council.deliberate() (5-role debate)
        |     else -> agentic_core.plan_and_run() (LLM JSON plan -> run steps)
        |               fallback -> llm_client.chat() (with history)
        v
  _remember_turn(): history[-40] + conversation_memory.log_turn()
                    (persisted to disk, secrets redacted)
        v
  reply -> channel -> voice (edge-tts neural / browser TTS)
```

3. Core Components

Layer	File	Role
Orchestrator	`core/orchestrator.py`	OODA pipeline, ~200-intent dispatch, single-flight `handle()`, persona assembly
Reasoning	`core/reasoning_core.py`	Rule-based intent classify (no LLM), offline `decompose()` planner
Agentic	`core/agentic_core.py`	LLM action planner (JSON `{say, steps}`) + ReAct `act_loop`; CATALOG of valid actions, MAX_STEPS=8
LLM	`core/llm_client.py`	Multi-brain (claude/perplexity/gemini/nvidia/ollama), failover, smart per-task routing, response cache, vision API
Query router	`core/query_router.py`	DIRECT (no search) / LIVE (time-sensitive) / DEEP (high-stakes)
Verification	`core/verification_council.py`	RESEARCH → ANSWER → CHALLENGE → VERIFY → SYNTHESIZE; surfaces dissent, never fabricates consensus
Council	`core/council.py` + `agents/council_agents.py`	FRIDAY (research), TRON (build/debug), ULTRON (red-team), IRA (security/veto), MYTHOS (creativity)

4. Permission & Safety Model

`permission_gate.classify(action, arg)` is the single source of truth. Phrasing/urgency can never downgrade a class.

CLASS A auto-run — read, search, compute, open apps, paper-trade, draft.
CLASS B confirm once — run_command, delete, shutdown, install, send, trade, spend, modify settings, change own config.
FORBIDDEN refuse — self-edit of the permission-enforcing files.

```
PROTECTED (never self-editable, even on command):
  permission_gate.py . safety_guard.py . permission_manager.py

Central rail: _run_action() routes ALL planner / agentic / autonomous actions
through the gate -> ONE enforcement point. A Class B action never auto-executes,
not under autonomy, urgency, or repeated instruction.
```

5. Self-Extension Pipeline

`_self_code` → `self_upgrade.py`. Trigger: "upgrade your code to X" / "modify your core to Y".

```
1. RESEARCH   -> Perplexity (pinned) / FRIDAY  : secure best-practice
2. DRAFT      -> Claude (pinned, no downgrade)  : writes the COMPLETE file
3. STAGE+TEST -> .sandbox_upgrade/              : compile + import validation (prod untouched)
4. SECURITY   -> IRA red-teams the diff         : injection, exec, weakened gates,
                 exfiltration, autonomy creep -> VERDICT APPROVE / CONCERNS / BLOCK
                 (FAILS CLOSED: no reviewer = BLOCK)
5. GATE       -> BLOCK = discard + no notify . else -> notify the master
6. APPROVE    -> "approve upgrade" -> backup -> overwrite -> log (self_upgrade.log)
                 undo: "rollback last upgrade"

Separate update paths: check_updates (git pull) . skill_install / toolkit_install (PyPI)
```

6. Memory

Store	File / path	Persistence
Facts / preferences	`memory_manager.py` → `memory/memory.json`	Categorized (user_profile, business…); secrets refused
Conversations	`conversation_memory.py` → `memory/conversations.jsonl`	Every turn; survives restart; seeds context on boot; searchable; secrets redacted
Episodes / lessons	`experience.py` → `memory/episodes.json`	Autonomy reflections
RAM history	`orchestrator.history[-40]`	Rolling; seeded from conversations on boot
LLM cache	`llm_client._cache`	300 s TTL; bypassed for live data

7. Tools & Agents

Device / system tools (`tools/`)

browser_root — CDP Chrome + auto-launch + fallback · windows_app — Start-Menu search launch · screen_control — capture (≤1568px for vision) · volume · power · network · input · process · trading · trading_bot · news · web · utility · youtube · email · file_ops · weather

Agents (`agents/` , via `agent_registry`)

FRIDAY — research · TRON — build/debug · ULTRON — red-team · IRA — security/veto · MYTHOS — creativity · + business: executive, marketing, ecommerce, customer_support, trading_research, coding, security…

Registries: `tool_registry.py` (config/tools.json) · `agent_registry.py` (config/agents.json).

8. Autonomy Layer

Component	File	Behavior
Iron Man Mode	`autonomy_daemon.py`	Works queued goals one at a time via MissionControl; day/step budgets; pauses for Class B confirm; reflects + banks lessons
Mission Control	`mission.py`	plan → act → verify loop; pauses on risky steps
Proactive Advisor	`proactive_advisor.py`	Watches current tab; volunteers ≤1 tip / gap; anti-Clippy; read-only
Planner	`planner.py`	Goal → step plan
Watch Manager	`watch_manager.py`	price / news / system / url monitors → alert buffer

All background daemons surface via the alert buffer (HUD banner / voice) and never bypass permission gates.

9. Config & Runtime

```
config/settings.json    - brains, voice, autonomy, remote, proactive, vision
config/permissions.json - tier framework
config/tools.json       - tool registry
config/agents.json      - agent registry
.env                    - API keys (ANTHROPIC / PERPLEXITY / NVIDIA / GEMINI); never stored elsewhere
start_my_chrome.bat     - (now optional) debug-Chrome launcher; browser_root auto-launches instead
main.py                 - default HUD (:8765) . --terminal for CLI
```

Persona: `JARVIS_PERSONA` + governance (Class A/B) + advisory + `AUTONOMY_FRAMEWORK.md` + memory/business blocks → system prompt.
