



Deep Technical Analysis — Antigravity

Workspace Template (EN)

Artifact: artifacts/analysis/deep_analysis_antigravity_2025-12-24_en.md

Date: 2025-12-24

Scope: Code-grounded architecture review + implemented changes (installer scripts + OpenSpec).

Table of Contents

- [Overview](#)
 - [Core philosophy: “zero-config” agent workspace](#)
 - [Technical architecture](#)
 - [Agent loop: Think → Act → Reflect](#)
 - [Memory: JSON history + incremental summarization](#)
 - [Local tools: dynamic discovery and execution](#)
 - [Context injection: `./context/*.md`](#)
 - [Swarm: Router-Worker multi-agent orchestration](#)
 - [MCP integration: multi-server, multi-transport tools](#)
 - [Ops: Docker + GitHub Actions CI](#)
 - [Implemented change: cross-platform installer scripts](#)
 - [Precision notes \(what's verified vs inferred\)](#)
 - [Conclusion](#)
-

Overview

This repository is a Python-first “agent workspace template” built around a **single, readable agent runtime** plus optional extensions:

- **Core agent runtime:** `src/agent.py` (`GeminiAgent`) — a loop that keeps memory, calls the Gemini API, and can optionally execute tools.
- **Persistent memory:** `src/memory.py` (`MemoryManager`) — JSON-backed chat history + a summary buffer.
- **Local tools:** `src/tools/*.py` — dynamically imported and registered at startup (no manual registry required).
- **Optional MCP integration:** `src/mcp_client.py` + `src/tools/mcp_tools.py` — connect to MCP servers and surface their tools as callables.
- **Multi-agent swarm:** `src/swarm.py` + `src/agents/*` — Router-Worker orchestration using specialist sub-agents.

Operationally, the repo is set up to be **easy to bootstrap** (installer scripts), **easy to run** (Docker/Docker Compose), and **testable** (pytest + GitHub Actions CI).

Core philosophy: “zero-config” agent workspace

The repo leans into “zero-config” in two concrete ways:

1. **Drop-in tools:** any public function inside `src/tools/*.py` becomes a tool without further wiring. The agent dynamically imports modules at runtime and registers their public functions.

2. **Drop-in context:** any Markdown file in `.context/*.md` gets concatenated and injected into the agent's system prompt during `think()` .

This keeps the learning curve low: the user doesn't have to learn a framework's DSL or config schema before getting something useful.

Technical architecture

Agent loop: Think → Act → Reflect

The main runtime is `GeminiAgent` in `src/agent.py` . Its loop is intentionally simple and readable:

(1) Think (`GeminiAgent.think(task)`)

- Loads context from `.context/*.md` via `_load_context()` .
- Builds a `system_prompt` that prepends the concatenated context and appends a short behavioral directive.
- Requests a bounded context window from `MemoryManager.get_context_window(..., max_messages=10, summarizer=self.summarize_memory)` .
- Prints a `<thought>` block to stdout (logging only; not a model primitive).

(2) Act (`GeminiAgent.act(task)`)

- Appends the user task to memory (`MemoryManager.add_entry("user", task)`).
- Builds a tool catalog string from registered callables (`_get_tool_descriptions()` pulls docstrings).
- Calls Gemini with a prompt that enforces one of two response formats:
 - a direct final answer, or
 - a JSON tool request: `{"action": "<tool_name>", "args": {...}}` .
- Extracts tool calls with `_extract_tool_call()` supporting:
 - JSON with `action` / `tool` and `args` / `input`
 - a plain-text `Action: <tool_name>` line
- If a tool is requested, executes **at most one tool** and then performs a follow-up model call including the tool observation while instructing the model not to request additional tools.

(3) Reflect (`GeminiAgent.reflect()`)

- Currently prints the history length only. There is no automated self-improvement step.

This design choice is important: it avoids “agent spaghetti” and makes the runtime easy to modify.

Memory: JSON history + incremental summarization

Memory is implemented in `src/memory.py` as `MemoryManager` .

Storage format

- A single JSON file (`settings.MEMORY_FILE` , default: `agent_memory.json`) containing:
 - `summary` (string)
 - `history` (list of message objects)

Context-window construction (`MemoryManager.get_context_window`)

- Always prepends a synthetic system message `{role: "system", content: system_prompt}` .

- If history length \leq `max_messages` , it returns system + full history.
- If history is longer, it:
 1. splits into `messages_to_summarize` (older) and `recent_history` (last `max_messages`)
 2. calls `summarizer(old_messages, previous_summary)`
 3. persists the merged summary back to disk
 4. returns: system message + `Previous Summary: ... message + recent history`

Summarizer implementation used by the agent

- `GeminiAgent.summarize_memory(...)` calls the Gemini model and requests a merged summary under **120 words**.
- This yields a practical “infinite-ish” conversation memory: older turns collapse into `summary` , while the newest turns remain verbatim.

Local tools: dynamic discovery and execution

Local tool discovery is in `GeminiAgent._load_tools()` :

- Scans `src/tools/*.py` .
- Skips private modules (`_* .py`) and registers public functions only.
- Imports modules dynamically via `importlib.util.spec_from_file_location(...)` .
- Registers a function only if:
 - its name does not start with `_` , and
 - it is defined in that module (`obj.__module__` check)

Tool invocation contract

- The agent asks the model to request a tool via JSON: `{"action": "tool_name", "args": {...}}` .
- Arguments are passed via `tool_fn(**tool_args)` .
- Output is recorded in memory as a `tool` role entry (`"<tool_name> output: <observation>"`).

Notable implication: the “schema” is implicit (docstrings + parameter names). There is no formal JSON schema validation layer in this agent loop.

Context injection: `.context/*.md`

Context injection is implemented in `GeminiAgent._load_context()` :

- Reads Markdown files in `.context/` **root** only (glob: `.context/*.md`), sorted by filename.
- Concatenates them into one string; each file is wrapped like:
 - `--- <filename> ---` followed by file content
- The resulting block is prepended to the system prompt inside `think()` .

This pattern is simple but effective for embedding:

- team coding standards
- domain notes
- repository conventions
- “do / don’t” operational policies

If recursive folder traversal is desired later, it would be a small extension (switching from `glob("*.md")` to `rglob("*.md")`).

Swarm: Router-Worker multi-agent orchestration

The multi-agent system is implemented in `src/swarm.py` and `src/agents/*`.

Components

- `SwarmOrchestrator` bootstraps:
 - a `RouterAgent` (manager/coordinator)
 - worker agents: `CoderAgent`, `ReviewerAgent`, `ResearcherAgent`
- `MessageBus` stores a chronological log of inter-agent messages with:
 - `from`, `to`, `type`, `content`, `timestamp`

Execution flow (`SwarmOrchestrator.execute(user_task)`)

1. Router produces a delegation plan via `RouterAgent.analyze_and_delegate()` .
 - It attempts to parse a structured plan from the Router output.
 - If parsing fails, it uses a keyword-based fallback (`_simple_delegate`).
2. For each delegation:
 - records the task on the message bus
 - retrieves contextual messages for that worker (`MessageBus.get_context_for`)
 - calls `worker.execute(agent_task, context)`
 - records the result back to the message bus
3. Router synthesizes the final answer via `RouterAgent.synthesize_results(delegations, results)` .

Important testing detail

- Each agent uses a dummy Gemini client under pytest (see `src/agent.py` and `src/agents/base_agent.py`), keeping tests deterministic without external network calls.

MCP integration: multi-server, multi-transport tools

MCP support is implemented in `src/mcp_client.py` and exposed via helper tools in `src/tools/mcp_tools.py`.

Configuration surface (`src/config.py`)

- `MCP_ENABLED` (default: `False`)
- `MCP_SERVERS_CONFIG` (default: `mcp_servers.json`)
- `MCP_TOOL_PREFIX` (default: `mcp_`)

Manager design (`MCPClientManager` in `src/mcp_client.py`)

- Loads enabled server entries from the JSON config.
- Connects to each server and discovers its tools.
- Supports transports:
 - `stdio` (spawns a command, pipes stdin/stdout)
 - `http` / `streamable-http`
 - `sse`

Tool naming

- Each MCP tool is represented as an `MCPTool` with:

- `server_name` , `original_name` , `description` , `input_schema`
- A prefixed tool name is constructed as:
 - `<prefix><server_name>_<original_name>`
 - e.g. `mcp_github_create_issue`

Sync wrapper for non-async environments

- `MCPClientManagerSync` wraps the async manager and exposes blocking methods (it creates/uses an event loop internally).
- The main agent (`GeminiAgent`) uses this sync wrapper to integrate MCP tools into the same `available_tools` dictionary as local tools.

User-facing MCP helper tools (`src/tools/mcp_tools.py`)

- `list_mcp_servers()`
- `list_mcp_tools(server_name=None)`
- `get_mcp_tool_help(tool_name)`
- `mcp_health_check()`

Ops: Docker + GitHub Actions CI

This repository includes a minimal but working ops story.

Dependencies (see `requirements.txt`)

- `google-genai` (Gemini client)
- `pydantic` + `pydantic-settings` (+ `.env` loading)
- `pytest` (tests)
- `requests` (OpenAI-compatible proxy tool)
- `mcp[cli]>=1.0.0` (optional MCP integration)

Docker

- The `Dockerfile` is a two-stage build on `python:3.12-slim` :
 - builder installs Python deps with `pip install --user`
 - runtime copies `/root/.local` into the final image
- Default container entrypoint runs: `python src/agent.py` .

Docker Compose

- `docker-compose.yml` builds the image, wires environment variables, and mounts `agent_memory.json` into the container for persistence.

CI (GitHub Actions)

- `.github/workflows/test.yml` runs `pytest tests/` on pushes and PRs targeting `main` .

Implemented change: cross-platform installer scripts

The repo includes onboarding scripts intended to remove setup friction:

- `install.sh` (Linux/macOS)
 - Checks `python3` presence and version (≥ 3.8)

- Checks `git`
- Creates `venv/` and installs `requirements.txt`
- Creates `.env` if missing and ensures `artifacts/` exists
- `install.bat` (Windows)
 - Checks `python` availability and `git`
 - Creates `venv\` and activates it
 - Installs `requirements.txt`
 - Creates `.env` and `artifacts\` if missing

This change was tracked and archived via OpenSpec:

- Archived change folder: `openspec/changes/archive/2025-12-24-add-installer-script/`
- Resulting spec: `openspec/specs/deployment/spec.md`

Precision notes (what's verified vs inferred)

Verified directly from repository contents

- Agent loop semantics, tool extraction format, and “one tool per iteration” behavior (`src/agent.py`).
- Memory storage format and summary-based context window (`src/memory.py`).
- Swarm architecture and delegation parsing/fallback (`src/swarm.py` , `src/agents/*`).
- MCP multi-transport manager design, sync wrapper, and tool naming (`src/mcp_client.py` , `src/tools/mcp_tools.py` , `src/config.py`).
- Docker/Compose and CI behavior (`Dockerfile` , `docker-compose.yml` , `.github/workflows/test.yml`).
- Installer scripts exist and implement setup flows (`install.sh` , `install.bat`).

Inferred / aspirational

- Any “policy enforcement” around artifacts, mandatory testing, or audit trails depends on the surrounding IDE/ops discipline. The Python runtime itself does not enforce those conventions.
- Performance/security comparisons against other frameworks are qualitative; they would require controlled benchmarks to quantify.

Conclusion

This template is best understood as a **minimal, inspectable agent runtime** plus a set of “workspace ergonomics”:

- The core agent loop is intentionally straightforward.
- Tooling is plug-and-play through dynamic discovery.
- Memory is pragmatic (JSON + summary buffer) and works well for iterative tasks.
- Optional MCP expands capabilities without complicating the local tool story.
- Swarm orchestration demonstrates how to scale from a single agent to specialist collaboration.

If you want, the next high-leverage improvement would be adding **formal tool schemas / validation** (so tool calls become more reliable), and optionally extending `.context` loading to support nested folders.