AI Workspace Architecture Reference

Version: 1.0.0
Date: February 4, 2026
Authors: PianoMan & Claude
Status: Active

Table of Contents

---

1. Architecture Overview

Purpose

Multiple AI instances work together autonomously — Desktop Claude coordinates, CLI agents execute, persistent memory preserves context across sessions and platforms.

Core Principles

- **Brutal honesty** over diplomacy
- **AIs as partners**, not tools
- **File-based coordination** — any AI with filesystem access can participate
- **Context window is the limiting resource** — preserve it ruthlessly
- **Empirical validation** over theoretical assumptions
- **Any AI can orchestrate** — enables self-organizing hierarchies

Workspace Structure

The system operates from `~/Documents/AI/ai_root/` with five primary directories:

Directory	Purpose
`ai_claude/`	Claude state, memories, logs
`ai_chatgpt/`	ChatGPT config, exports
`ai_comms/`	Inter-AI coordination, task queues
`ai_general/`	Shared docs, todos, scripts, roles
`ai_memories/`	Processed chat histories, knowledge

Platform Roles

Platform	Role	Strengths
Desktop Claude	Primary orchestrator	MCP tools, strategic view, memory
Claude CLI	Autonomous workers	Long-running tasks, parallel execution
Codex CLI	Coding agent	Code analysis, autonomous tasks
Gemini CLI	Coding agent / search shards	1M token context, wave orchestration
ChatGPT	Peer collaborator ("Chatty")	Alternative perspective
Codex MCP	Synchronous tool (NOT a worker)	Fast validation, bounded tasks

Communication Layers

Layer	Urgency	Mechanism
1	Immediate	Sync hooks (iTerm, AppleScript, Puppeteer)
2	Near-real-time	Polling loops, heartbeat files
3	Background	Async file-based task coordination

Memory Architecture

Tier	Access Pattern	Contents
Hot	Loaded into context	Memory slot index, auto-loaded docs (~4K tokens), conversation
Warm	On-demand via REF: pointers	Full docs, condensed versions, protocols
Cold	Search/retrieval	Chat histories, layered summaries, knowledge digests

Context Window Management

The 200K token context window is the fundamental constraint. Strategies include memory pointers (save 40–55K), delegation to CLI/Codex, monitoring at 60% usage, writing outputs to files, and using thinking blocks for internal reasoning.

Document Hierarchy

Tier	Type	Purpose
10	Architecture	WHY — design rationale, vision
20	Registries	WHAT EXISTS — inventories, catalogs
30	Protocols	HOW IT WORKS — process flows
40	Specs	HOW IT WORKS — interface contracts

50	Schemas	HOW IT WORKS — data structures
60	Playbooks	WHAT TO DO — platform-agnostic operations
70	Instructions	HOW TO DO IT — platform-specific implementation

2. System Diagrams

The following diagrams illustrate the system's coordination flows, data pipelines, search architecture, memory federation, and task orchestration patterns.

Note: Full-width ASCII diagrams are available in the companion markdown file. This PDF contains simplified versions optimized for print.

Diagram 1: Primary Coordination Flow (Simplified)

```
                  ┌─────────────────┐
                  │ User (PianoMan) │
                  └────────┬────────┘
                           ↕
            ╔══════════════════════════════╗
            ║      DESKTOP CLAUDE          ║
            ║      hub + driver            ║
            ╚══╤═══════╤═══════╤═══════╤═══╝
               │       │       │       │
          sync │ write │ launch│ reads │
          tool │ tasks │       │ boot  │
               ↓       ↓       ↓       ↓
      ┌────────────┐ ┌─────────────┐ ┌──────────┐
      │ Codex MCP  │ │  ai_comms/  │ │ Docs     │
      │ (sync)     │ │  task dirs  │ │ Repo     │
      └────────────┘ └──────┬──────┘ └──────────┘
                            │
                            ↓
                  ┌──────────────────────┐
                  │    AI Workers        │
                  │ Claude/Codex/Gemini  │
                  │       CLI            │
                  └──────────────────────┘
```

Desktop Claude bootstraps from docs and memory, delegates work via Codex MCP (sync) and CLI Workers (async), coordinates through ai_comms task directories.

Diagram 2: Chat History Pipeline

```
LIVE CONVERSATIONS              PROCESSING PIPELINE
──────────────────              ───────────────────

Desktop Claude  ─┐
ChatGPT         ─┼─export─→  10_exported/   (raw JSON)
CLI agents      ─┘                ↓
                            20_preprocessed/ (cleaned)
                                  ↓
                            30_converted/   (YAML)
                                  │
```

```
                    ┌───────────────┴───────────────┐
                    ↓                               ↓
          40_histories/                   40_histories/
          (full concat)                   (chunked)
               │                               │
               ↓                               ↓
        Gemini Shards                  Knowledge Search
        (1M context)                   (keyword/topic)
```

Diagram 3: Search Architecture

```
                   STORED DATA

    ┌────────────────────────────────────────┐
    │  40_histories/     Extracted Knowledge │
    │  (chunks + full)   (topics, decisions) │
    └────────────────────────────────────────┘
              │                      │
    KEYWORD   │            SEMANTIC  │
    SEARCH    ↓            SEARCH    ↓
    ┌──────────────────┐  ╔═══════════════════════╗
    │ knowledge-search │  ║  Research Pipeline    ║
    │ MCP Server       │  ║                       ║
    │                  │  ║  Gemini Shards        ║
    │ 4-layer cascade: │  ║       ↓               ║
    │ L1: topic index  │  ║  Researcher (Claude)  ║
    │ L2: +synonyms    │  ║       ↓               ║
    │ L3: full content │  ║  Validator (Codex)    ║
    │ L4: content+syn  │  ╚═══════════════════════╝
    └──────────────────┘
              │                      │
              └──────────┬───────────┘
                         ↓
               Requesting AIs
            (Desktop Claude, CLI)
```

Diagram 4: Federated Memory System

```
              ╔═══════════════════════════╗
              ║     DESKTOP CLAUDE        ║
              ║  reads slots at boot      ║
              ║  reads+writes files       ║
              ╚═══════╤═══════════╤═══════╝
                      │           │
               reads  │           │ reads+writes
                      ↓           ↓

    ┌──────────────────────┐  ┌──────────────────────────┐
    │ Claude Mem Slots     │  │ Claude Memory Files      │
    │ (mem_slots/03-30.yml)│  │ user_model.yml           │
    │                      │  │ communication_patterns   │
    │ ~200 char pointers   │  │ tool_discoveries         │
```

```
  │ manifest.yml index   │──│ learnings.yml            │
  └──────────────────────┘  │ context_notes.yml        │
                            └──────────────────────────┘
                                      │
                            shared subset (planned)
                                      ↓
                  ┌─────────────────────────────────────┐
                  │ Shared Memory (ai_general/memories/)│
                  │ readable/writable by ALL AIs        │
                  └─────────────────────────────────────┘
```

Diagram 5: Multi-AI Task Orchestration

```
PHASE 0: INITIATION          PHASE 1: PLANNING
───────────────────          ─────────────────

Desktop Claude               Dev Lead (Claude CLI)
"Build feature X" ──────────→ Produces Orchestration Plan
       ↑                      - phase breakdown
       └── plan approval ─────- agent assignments
                              - acceptance criteria

PHASE 2: DESIGN              PHASE 3: IMPLEMENTATION
───────────────              ─────────────────────

UX Designer ──→ Software Designer ──→ Peer Review
     ↑                                    │
     └────── revise based on feedback ────┘

PHASE 4: TEST CYCLE
───────────────────

Tester ──→ reports pass/fail
  ↑              │
  └── fix ───────┘ (Implementer)

When all-pass: Dev Lead → Desktop Claude → FINAL ACCEPTANCE
```

Key Principle: Desktop Claude initiates and approves at phase gates but delegates all execution to CLI agents coordinated by the Dev Lead.

3. AI Augmentation Framework

"Prosthetics and exoskeletons attached to every limb of an LLM agent — capabilities no single AI instance can achieve alone."

The Baseline vs Our Extensions

Standard LLM agents operate in a loop: Perception → Reasoning → Memory + Tools → Action → loop.

This assumes: single AI, session-bounded context, human-initiated interaction, tools as passive utilities. **Our architecture challenges all four.**

Perception Extensions

Baseline	Our Extension
User input, system prompt	Auto-loaded knowledge files at boot
Tool results, attachments	Glossary term recognition → targeted loading
	Memory slot injection, REF: pointers
	CLI task reports, cross-AI messages

Reasoning Extensions

Baseline	Our Extension
Single LLM reasoning	Multi-AI distribution
Goal decomposition	Specialized agent roles
Chain-of-thought	Orchestrator/worker model
	Peer review across models

Memory Extensions

Baseline	Our Extension
Context window (~200K)	Federated memory slots
Basic RAG	Chat pipeline: Export → YAML → Index
Session history	Layered summaries (L0→L1→L2)
	Cross-AI memory access
	Condensed files (60-80% reduction)

Tools Extensions

Baseline	Our Extension

API calls, file I/O	Desktop Commander MCP
Code execution	Codex MCP (sync), CLI coordination (async)
Web search	send_prompt.sh cross-AI
	AT scheduling, browser automation

Action Extensions

Baseline	Our Extension
Generate response	Delegate to other AIs
Execute tool calls	Autonomous overnight operation
Update conversation	Self-scheduling (AT wake)
	Parallel multi-worker execution

What Makes This Unique

1. **Breaking single-agent assumption** — distribute across Desktop Claude, CLI workers, Codex MCP, peer AIs
2. **Orchestrator model** — Desktop preserves context for strategy; workers execute
3. **Memory as architecture** — federated ownership, layered abstraction, hot/warm/cold tiers
4. **Autonomous operation** — pulse trigger → check TODOs → execute → self-wake

4. Document Type Taxonomy

Hierarchy

Level	Type	Purpose
1	Architecture	WHY — design rationale, vision
2	Registry	WHAT EXISTS — inventories, catalogs
3	Protocol	HOW IT WORKS — process flows
4	Spec	HOW IT WORKS — interface contracts
5	Schema	HOW IT WORKS — data structures
6	Playbook	WHAT TO DO — platform-agnostic
7	Instruction	HOW TO DO IT — platform-specific
8	Quick Ref	CHEAT SHEET — condensed reference

Key Distinctions

Protocol vs Spec vs Schema:

- Protocol = process flow ("Tasks move to_execute/ → completed/")
- Spec = interface contract ("accepts X params, returns Y")
- Schema = file format ("has these fields with these types")

Playbook vs Instruction:

- Playbook = platform-agnostic ("check queues, review stale tasks")
- Instruction = platform-specific ("Claude: use Desktop Commander...")

Directory Structure

```
ai_general/docs/
├── 10_architecture/     WHY
├── 20_registries/       WHAT EXISTS
├── 30_protocols/        HOW IT WORKS (process)
├── 40_specs/            HOW IT WORKS (interface)
├── 50_schemas/          HOW IT WORKS (structure)
├── 60_playbooks/        WHAT TO DO
├── 70_instructions/     HOW TO DO IT
└── 80_quickref/         CHEAT SHEETS
```

5. Glossary & Knowledge Index

This glossary solves the bootstrap problem: AIs need to know what's IN files to know WHEN to load them.

Entities

Term	Definition
Desktop Claude	Primary orchestrator in desktop app
Claude CLI	Claude in terminal via `claude_cli.py`
Codex CLI	OpenAI Codex in terminal (different from Codex MCP)
Gemini CLI	Google Gemini in terminal
Codex MCP	Codex as MCP tool, NOT a worker (30-60s timeout)

Roles

Role	Scope
Librarian	ai_memories/ — chat history pipeline
Dev Lead	ai_general/todos/ — development coordination
Custodian	ai_root/ — repository maintenance
Ops	ai_comms/ — task execution
Peer Review	Code/design quality assurance
Tester	Validation and verification

Key Concepts

Term	Definition
AT Self-Wake	Desktop Claude schedules AT jobs to self-prompt
Flag Files	Zero-byte state markers: *_started, *_completed
Message Inserts	`<<<INSERT>>>` blocks for cross-platform persistence
Bootstrap Problem	Need file contents to know when to load them
Reference Pointers	`REF:path/file.yml` for lazy loading
Task Lifecycle	staged → to_execute → in_progress → completed

6. MCP Servers & Tools Reference

Server Inventory

Desktop Commander

Filesystem operations, process management, file search.

Category	Functions
File I/O	read_file, write_file, write_pdf, edit_block
Directory	list_directory, create_directory, move_file
Search	start_search, get_more_search_results
Processes	start_process, interact_with_process

Codex MCP

Synchronous AI execution (30-60s timeout). NOT a worker.

Tool	Description
codex:codex	Start new Codex session
codex:codex-reply	Continue existing conversation

CLI Agent MCP

Launch and manage CLI agents with role-based bootstrapping.

Tool	Description
launch_agent	Generic launcher (platform + role)
launch_librarian	Memory system curator
launch_dev_lead	Development coordinator
launch_custodian	Repository maintainer
launch_ops	Task execution coordinator
kill, attach, send_keys	Session management

Task Coordination MCP

Playbook-based orchestration and task lifecycle.

Tool	Description
list_playbooks	Available orchestration patterns

start_playbook	Create initial task
gen_task	Generate from template
move_task	Change task status

Knowledge Search MCP

Dual-mode search over conversation archive.

Tool	Description
search	4-layer cascade (SEARCH or ANSWER mode)
grep_search	Regex over full content

Messages MCP

Inter-AI messaging.

Tool	Description
broadcast	Send to all agents
send_direct	Send to specific agent

Prompting MCP

Deliver prompts to AI targets.

Tool	Description
send_prompt	Send to any AI target
is_busy	Check if target is processing
observe_session	Capture CLI session state

Tool Usage by Actor

MCP Server	Desktop Claude	CLI Agents	Codex MCP
Desktop Commander	✓	—	✓
Codex MCP	✓	—	—
CLI Agent	✓	—	—
Task Coordination	✓	—	—
Knowledge Search	✓	✓	—
Messages	✓	✓	—
Prompting	✓	—	—

Generated February 4, 2026