3. The Pipeline

BUILD → REVIEW → SECURITY → OPS → DOCS

3.1 Pipeline Stages

Pipeline Stages

The Pipeline Engine (`~/system/kernel/pipeline-engine.js`) manages a multi-company chain where each completed stage triggers the next.

Stage Flow

```
BUILD (CodeCraft / Vizu / Datavera)
  │
  ▼  Task completed → Pipeline Engine creates REVIEW task
REVIEW (Proveo)
  │
  ├── Pass    → Pipeline Engine creates SECURITY task
  └── Reject  → New BUILD task with findings (max 2 cycles)
        │
        ▼
SECURITY (Securion)
  │
  ├── Pass    → Pipeline Engine creates OPS task (if deploy)
  └── Reject  → New BUILD task with security findings
        │
        ▼
OPS (FlowForge) — optional, triggered for deploy tasks
  │
  ▼
DOCS (Lexicon) — optional, triggered for docs tasks
```

How It Works Technically

1. **Pi-Orchestrator** completes a BUILD task
2. Calls `pipeline-engine.js advance <task-id>`
3. Pipeline Engine checks the task's pipeline stage

4. Creates a new MC task for the next stage with:
   - Company context injected
   - Previous stage output as input
   - Read-only flag for REVIEW/SECURITY (they can't modify code)
5. Pi-Orchestrator picks up the new task in the next cycle

Pipeline Metadata

Each task in a pipeline has metadata tracking:

- `pipeline_id` — groups related tasks
- `pipeline_stage` — current stage (BUILD/REVIEW/SECURITY/OPS/DOCS)
- `pipeline_parent` — ID of the task that triggered this one
- `review_cycle` — current review iteration (max 2)

Safety: Max Review Cycles

To prevent infinite BUILD↔REVIEW loops, the pipeline caps at **2 review cycles**. After that, the task is queued for human review.

3.2 Pi-Orchestrator

Pi-Orchestrator

The Pi-Orchestrator (`~/system/kernel/pi-orchestrator.js`) is the **central brain** of the ALAI system. It's a daemon that runs 24/7.

What It Does

Every **30 seconds**, the orchestrator:

1. **Polls MC** for the next eligible task (`mc.js next-task`)
2. **Skips** tasks matching safety patterns (CEO decisions, regulated services)
3. **Checks retry cap** — max 3 attempts per task (prevents infinite loops)
4. **Classifies** the task (complexity 1-5, domain, type) using Ollama
5. **Selects model** based on classification tier
6. **Checks capacity** — worker pool limits per host
7. **Claims the task** (assigns to pi-orchestrator, sets status to started)
8. **Builds prompt** — injects company soul, blueprints, task context
9. **Executes** via the selected pipeline (ollama-tool → simple fallback → llama-server)
10. **Quality gate** — checks response length, detects placeholders, escalates if needed
11. **Creates proof-of-work** — writes GOTCHA file + verify dir
12. **Completes** — marks task done in MC, posts to Slack, feeds HiveMind
13. **Advances pipeline** — triggers next pipeline stage if applicable

Configuration

`~/system/config/pi-orchestrator-config.json`:

- **Concurrency:** FORGE: 2, ANVIL: 3, Claude: 2 (total max: 6 parallel workers)
- **Timeouts:** Tier 1: 60s, Tier 2: 120s, Tier 3: 300s, Tier 4: 600s
- **Quality gate:** Min response lengths, placeholder detection, max 1 escalation
- **Safety patterns:** CEO, DECISION, BankID, Vipps, Finanstilsynet, TENDER

Execution Pipelines

Pipeline	How It Works

`ollama-tool`	Ollama with tool-use (read/write files, run commands) — preferred
`ollama-simple`	Ollama text-only (fallback when tool agent fails)
`llama-tool`	Kimi K2.5 via llama-server with tool-use wrapper
`llama-server`	Kimi K2.5 raw text completion (fallback)
`claude-cli`	Claude Code CLI for client-facing/critical work
`human-queue`	Queued for human — Alem reviews manually