ALAI System Architecture — How It All Works

Complete documentation of the ALAI AI Factory: virtual companies, pipeline, agents (minions), orchestration — with a concrete use case for creating a new product.

1. System Overview

The big picture — what ALAI is and how the pieces fit together

1.1 What is ALAI?

What is ALAI?

ALAI Holding AS is an **AI-first company factory**. Instead of a traditional software company with developers, ALAI operates a system of AI agents organized into virtual companies that build, review, secure, and deploy software products autonomously.

Core Principle

> "ALAI AS is not a holding company that happens to have a build pipeline. ALAI AS **IS** the build pipeline. The pipeline is the product."

The Machine

The system runs on two Mac Studio M3 Ultra machines:

Machine	Codename	RAM	Role
Mac Studio 1	**ANVIL**	96 GB	Infrastructure — databases, Docker, daemons, orchestration, Cloudflare tunnels
Mac Studio 2	**FORGE**	256 GB	Compute — AI inference, heavy models (72B+), agent worker pool

Connected via 10Gbps Thunderbolt bridge (10.0.0.1 ↔ 10.0.0.2).

Key Components

```
ALAI System
│
├── Mission Control (MC)          — Task database (SQLite) — the single source of truth
├── Pi-Orchestrator               — Daemon that pulls tasks from MC, classifies, routes,
executes
├── Virtual Companies (8 active)  — Specialized AI agent teams with souls, configs, blueprints
├── Pipeline Engine               — BUILD → REVIEW → SECURITY → OPS → DOCS chain
├── Minion System                 — One-shot autonomous agents for task execution
├── HiveMind                      — Shared knowledge base with semantic search (Qdrant)
├── Ollama Fleet                  — Multi-host model serving (qwen, kimi, llama, custom
models)
└── BookStack                     — Documentation wiki (this!)
```

Products Currently Built by the System

Product	Description	Status
Drop	Digital banking / remittance platform (PSD2-compliant)	Active development
Plock	AI-native warehouse management system (Sweden)	Active development
Lobby	AI-native HR platform	Active development
Bilko	Balkan accounting software	Active development
BasicFakta	Invoice/faktura tool	Active development
Tok	Open banking platform	Active development

1.2 How Everything Connects

How Everything Connects

The Flow: From Idea to Done

```
CEO Decision / Task Creation
        │
        ▼
   Mission Control (MC)          ← Task enters the system
        │
        ▼
   Pi-Orchestrator               ← Daemon picks it up every 30s
        │
        ├── 1. Classify           (complexity, domain, type)
        ├── 2. Route              (select model + company)
        ├── 3. Build Prompt       (inject company soul, blueprints, context)
        ├── 4. Execute            (via ollama-tool, llama-server, or claude)
        ├── 5. Quality Gate       (verify output, escalate if needed)
        └── 6. Complete           (mark done, advance pipeline, feed HiveMind)
                │
                ▼
         Pipeline Engine          ← Triggers next stage
                │
                ├── BUILD done   → Create REVIEW task (Proveo)
                ├── REVIEW done  → Create SECURITY task (Securion)
                ├── SECURITY done → Create OPS task (FlowForge) if deploy needed
                └── OPS done     → Create DOCS task (Lexicon) if docs needed
```

Model Tier System

Tasks are classified by complexity (1-5) and routed to the appropriate AI model:

Tier	Model	Host	Pipeline	Use Case
1	qwen3:8b	FORGE	ollama-tool	Simple tasks, classification
2	qwen2.5-coder:32b	ANVIL	ollama-tool → ollama-simple	Medium complexity code
3	Kimi K2.5 (240GB)	FORGE	llama-tool → llama-server	Complex tasks, architecture
4	Claude Sonnet	Cloud	claude-cli	Client-facing, critical work
5	Human	—	human-queue	CEO decisions, sensitive ops

Safety Rails

Tasks matching these patterns are **never auto-processed**:

- `CEO` / `DECISION` — requires Alem's input
- `BankID` / `Vipps` / `Finanstilsynet` — regulated services
- `[TENDER]` — procurement (currently paused)

2. Virtual Companies

The 8 active AI companies and how they specialize

2.1 Company Overview

Virtual Companies

Virtual companies are **specialized AI agent teams**. Each company has:

- A **SOUL** (identity, values, expertise description)
- A **config** (domain, model preferences, agent roles)
- **Blueprints** (templates for how to build things)
- **Agents** (lead, builder, reviewer — each with model assignments)

Active Companies

Company	Type	Domain	Pipeline Role
CodeCraft	Dev Shop	Backend, APIs, databases, full-stack	BUILD (backend)
Vizu	Agency	Frontend, UI/UX, design, components	BUILD (frontend)
Datavera	Product Co.	Data engineering, analytics, ML, reporting	BUILD (data)
Proveo	Audit Firm	QA, testing, code review, validation	REVIEW
Securion	Consultancy	Security auditing, pen testing, threat modeling	SECURITY
FlowForge	Consultancy	DevOps, infra, CI/CD, deployment	OPS
Lexicon	Consultancy	Legal docs, compliance, GDPR, ToS	DOCS / LEGAL
AgentForge	AI Lab	Agent development, AI tooling, model fine-tuning	R&D

Merged / Archived Companies

Company	Merged Into	Reason
Skybound	CodeCraft	Overlap with backend dev
Proxima	Vizu	Overlap with frontend
Skillforge	FlowForge	Overlap with DevOps
HelixSupport	FlowForge	Overlap with ops
Finverge	Lexicon	Overlap with compliance

Company File Structure

```
~/companies/CodeCraft/
├── company.json     — Registration (name, type, org number)
├── config.json      — Domain, models, agent definitions
├── CLAUDE.md        — Soul/instructions for AI agents working as this company
├── README.md        — Company description
├── agents/          — Agent-specific prompts and configs
├── blueprints/      — Project templates (nextjs-app.yaml, api-backend.yaml)
├── logs/            — Execution logs
└── state/           — Current work state
```

Routing: Which Company Gets the Task?

The pi-orchestrator classifies each task and routes it:

```
Task title/description
       │
       ▼
  Keyword matching (routing.json)
       │
       ├── "frontend", "UI", "react"     → Vizu
       ├── "data", "analytics", "ML"     → Datavera
       ├── "security", "audit", "CVE"    → Securion
       ├── "deploy", "CI/CD", "infra"    → FlowForge
       ├── "legal", "GDPR", "ToS"        → Lexicon
       └── default (backend, API, etc.)   → CodeCraft
```

2.2 Company Agent Roles

Company Agent Roles

Each company has three agent roles:

Lead Agent (Orchestrator)

- **Model:** Typically Claude Sonnet or Kimi K2.5
- **Role:** Breaks down tasks, creates execution plans, coordinates
- **When used:** Complex tasks (complexity ≥ 3)

Builder Agent (Executor)

- **Model:** qwen2.5-coder:32b or Claude Sonnet
- **Role:** Writes code, creates files, implements features
- **When used:** All BUILD tasks

Reviewer Agent (Validator)

- **Model:** Same tier or higher than builder
- **Role:** Reviews output, checks quality, suggests improvements
- **When used:** REVIEW pipeline stage (Proveo), or internal quality gate

Agent Configuration Example (CodeCraft)

```
{
  "agents": {
    "lead": {
      "model": "sonnet",
```

```
      "type": "orchestrator"
    },
    "builder": {
      "model": "sonnet",
      "type": "builder"
    },
    "reviewer": {
      "model": "sonnet",
      "type": "reviewer"
    }
  },
  "tier_overrides": {
    "lint": "ollama:qwen3:8b",
    "classify": "ollama:qwen3:8b",
    "code_review": "ollama:qwen2.5-coder:32b",
    "debug": "ollama:qwen2.5-coder:32b",
    "build": "sonnet",
    "architecture": "opus",
    "database_design": "ollama:deepseek-r1:32b"
  }
}
```

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

4. Agents & Minions

Autonomous task execution agents

4.1 Minion System

Minion System

Minions are **one-shot autonomous agents** (inspired by Stripe's internal system). Each minion:

1. Gets a single task
2. Runs in an isolated git worktree
3. Has full tool access (read/write files, run commands)
4. Follows a blueprint (YAML chain of steps)
5. Produces verifiable output
6. Exits

How to Use

```
# Run a minion for a specific task
node minion.js run "Fix login validation bug" --project ~/ALAI/products/Drop

# Run a minion tied to an MC task
node minion.js run "Add rate limiting" --project ~/ALAI/products/Drop --mc-task 1234

# Queue for batch processing
node minion.js enqueue "Refactor auth module" --project ~/ALAI/products/Drop
```

Minion Execution Flow

```
Task + Project
     |
     ▼
1. Create isolated git worktree (branch: minion/<id>)
2. Load context (project README, relevant files, HiveMind knowledge)
3. Select blueprint based on task type
4. Execute blueprint steps:
```

```
    │
    ├── Step 1: Analyze (understand the codebase)
    ├── Step 2: Plan (create execution plan)
    ├── Step 3: Build (implement changes)
    ├── Step 4: Test (run tests, verify)
    └── Step 5: Report (write GOTCHA summary)
    │
5. Quality gate — verify output, run tests
6. Return result (success/failure + artifacts)
```

Blueprint Types

Blueprint	Purpose
`minion-one-shot.yaml`	General purpose — analyze, plan, build, test
`minion-bugfix.yaml`	Bug fixing — reproduce, diagnose, fix, verify
`minion-refactor.yaml`	Refactoring — understand, plan, refactor, test
`minion-security-fix.yaml`	Security fixes — audit, fix, verify
`minion-docs.yaml`	Documentation — read code, generate docs
`codecraft-nextjs-app.yaml`	Full Next.js app scaffold (CodeCraft)
`codecraft-api-backend.yaml`	API backend scaffold (CodeCraft)
`securion-security-review.yaml`	Security audit chain (Securion)

Git Worktree Isolation

Each minion runs in its own git branch/worktree:

- **Branch:** `minion/<short-id>`
- **Path:** `~/system/.claude/worktrees/minion-<id>`
- Changes don't affect main branch until merged
- Multiple minions can work in parallel on different branches

4.2 HiveMind — Shared Knowledge

HiveMind — Shared Knowledge Base

HiveMind (`~/system/agents/hivemind/hivemind.js`) is the **collective memory** of all agents.

What Goes Into HiveMind

- **Task completions** — every pi-orchestrator task result (Tier 2+)
- **Agent discoveries** — facts, patterns, warnings found during work
- **Session summaries** — condensed logs of work sessions
- **Manual knowledge** — docs, runbooks, architecture decisions

How It Works

```
Agent completes task
       |
       ▼
HiveMind stores entry:
  - Text (up to 5000 chars)
  - Source (which agent/engine)
  - Type (knowledge, alert, briefing, update)
  - Timestamp
       |
       ▼
BGE-M3 embeds the entry → Qdrant vector DB
       |
       ▼
```

```
Future agents query HiveMind:
  "What do we know about Drop's payment system?"
       |
       ▼
Semantic search → relevant entries returned as context
```

Usage

```
# Query knowledge
node hivemind.js query "How does Drop handle PSD2 consent?"

# Store new knowledge
node hivemind.js post my-agent knowledge "Drop uses SCA for all transactions over 30 EUR"

# Search by tag
node hivemind.js search --type alert --since 24h
```

Why It Matters

Without HiveMind, every agent starts from zero. With HiveMind:

- Agents learn from previous work
- Duplicate investigation is avoided
- Institutional knowledge persists across sessions
- Quality improves over time (flywheel effect)

5. Use Case: Creating a New Product

Step-by-step walkthrough of how the system builds a new product from scratch

5.1 Scenario: Building "Bento" — a Meal Prep SaaS

Use Case: Building "Bento" — a Meal Prep SaaS

Let's walk through how the ALAI system would build a completely new product from CEO idea to deployed MVP.

The Idea

> Alem: "We need a meal prep subscription SaaS for the Nordic market. Next.js frontend, Kotlin/Ktor backend, PostgreSQL. Call it Bento."

Step 1: Task Creation in Mission Control

Alem (or John) creates the initial tasks:

```
node mc.js add "Bento: Project scaffold — Next.js frontend + Kotlin/Ktor API" -p H -t alem
node mc.js add "Bento: Database schema design — users, subscriptions, meals, deliveries" -p H
node mc.js add "Bento: Landing page design — Nordic minimalist, hero + pricing + CTA" -p H
node mc.js add "Bento: API — user auth + subscription management endpoints" -p H
node mc.js add "Bento: Frontend — meal selection + checkout flow" -p H
```

These 5 tasks enter MC as `[open]` status.

Step 2: Pi-Orchestrator Picks Up Tasks

Within 30 seconds, the orchestrator wakes up:

```
[INFO] Found task #4001: Bento: Project scaffold — Next.js frontend + Kotlin/Ktor API
[INFO] Task #4001 classified: complexity=3 domain=code type=scaffold
[INFO] Task #4001 → llama-tool (kimi-k2.5:tq1 on forge)
```

Classification logic:

- "Project scaffold" + "Next.js" + "Kotlin/Ktor" → domain: code, type: scaffold, complexity: 3
- Complexity 3 → Tier 3 model (Kimi K2.5 on FORGE)

Routing:

- "Next.js frontend" → keyword match → **CodeCraft** (fullstack covers both)
- CodeCraft's soul + blueprints are injected into the prompt

Step 3: CodeCraft Builds It

The agent receives a prompt containing:

1. CodeCraft's company identity and coding standards
2. The `nextjs-app.yaml` blueprint template
3. The task description
4. Relevant HiveMind context (if any previous Bento-related work exists)

Agent output:

- Project structure created
- `package.json`, `docker-compose.yml`, `README.md`
- Basic Next.js pages + Kotlin API skeleton
- GOTCHA file written as proof of work

```
[INFO] Auto QA prep completed for task #4001
[INFO] Task #4001 completed via llama-server/kimi-k2.5:tq1 (912s)
```

5.2 Pipeline Cascade: Review ? Security ? Deploy

Pipeline Cascade

Step 4: Pipeline Triggers REVIEW

After BUILD completes, the Pipeline Engine fires:

```
pipeline-engine.js advance 4001
  → Stage: BUILD complete
  → Creating REVIEW task for Proveo
```

New MC task created:

```
#4006 [M] [open] [—] [BENTO-REVIEW] Code review of project scaffold (parent: #4001)
```

Pi-Orchestrator picks it up → routes to **Proveo** (audit firm):

Proveo's agent reviews:

- □ Project structure follows conventions
- ⚠ Missing .env.example
- ⚠ No health check endpoint
- □ Docker setup correct

Result: REVIEW passes with minor findings.

Step 5: Pipeline Triggers SECURITY

```
pipeline-engine.js advance 4006
  → Stage: REVIEW complete (pass)
  → Creating SECURITY task for Securion
```

New MC task:

```
#4007 [M] [open] [—] [BENTO-SECURITY] Security audit of project scaffold (parent: #4006)
```

Securion's agent checks:

- □ No hardcoded secrets
- □ CORS configured correctly
- ⚠ CSP headers missing
- ⚠ Rate limiting not implemented

Result: SECURITY passes with findings → creates follow-up BUILD task for fixes.

Step 6: Parallel Processing

While the scaffold goes through the pipeline, the other tasks are also being processed:

```
[22:05] Task #4002 (DB schema)    → CodeCraft → Kimi K2.5
[22:08] Task #4003 (Landing page) → Vizu      → qwen2.5-coder:32b
[22:10] Task #4004 (Auth API)     → CodeCraft → Kimi K2.5
[22:13] Task #4005 (Meal UI)      → Vizu      → qwen2.5-coder:32b
```

Note: Vizu handles the frontend (landing page, meal UI), CodeCraft handles the backend (DB, API). Each gets their company-specific soul and blueprints.

Step 7: OPS Stage (Deploy)

If a task is tagged for deployment:

```
pipeline-engine.js advance 4007
  → Stage: SECURITY complete
  → Task has deploy trigger
  → Creating OPS task for FlowForge
```

FlowForge creates:

- Docker build pipeline
- GitHub Actions CI/CD
- Staging environment config
- Health check monitoring

Step 8: DOCS Stage

Finally, Lexicon creates documentation:

- Privacy Policy (GDPR-compliant for Nordic market)
- Terms of Service
- API documentation
- User guides

5.3 End-to-End Timeline

End-to-End Timeline

What the System Produces

Starting from 5 MC tasks, the full pipeline generates:

Stage	Tasks	Company	Time
BUILD — scaffold	1	CodeCraft	~15 min
BUILD — DB schema	1	CodeCraft	~10 min
BUILD — landing page	1	Vizu	~8 min
BUILD — auth API	1	CodeCraft	~15 min
BUILD — meal UI	1	Vizu	~10 min
REVIEW (all 5)	5	Proveo	~5 min each
SECURITY (all 5)	5	Securion	~5 min each
BUILD — security fixes	2-3	CodeCraft	~10 min each
OPS — deploy setup	1	FlowForge	~10 min
DOCS — legal + API docs	2	Lexicon	~8 min each

Total: ~20-25 tasks auto-generated from 5 initial tasks.

Elapsed time: ~3-4 hours (tasks run sequentially on each host, some parallel on ANVIL+FORGE).

Human involvement: Zero (unless safety patterns trigger).

What Alem Sees

```
$ node mc.js list --tag bento

#4001  [H] [done] Bento: Project scaffold
```

```
#4002  [H] [done] Bento: Database schema design
#4003  [H] [done] Bento: Landing page design
#4004  [H] [done] Bento: API — auth + subscriptions
#4005  [H] [done] Bento: Frontend — meal selection + checkout
#4006  [M] [done] [BENTO-REVIEW] Code review: scaffold
#4007  [M] [done] [BENTO-SECURITY] Security audit: scaffold
#4008  [M] [done] [BENTO-REVIEW] Code review: DB schema
...
#4020  [M] [done] [BENTO-OPS] Deploy setup
#4021  [M] [done] [BENTO-DOCS] Privacy Policy + ToS
#4022  [M] [done] [BENTO-DOCS] API documentation
```

Key Insight

The 5 tasks Alem created cascaded into 20+ tasks that were:

- **Automatically created** by the Pipeline Engine
- **Automatically classified** by the Pi-Orchestrator
- **Automatically routed** to the right virtual company
- **Automatically executed** by AI agents
- **Automatically quality-checked** by the QA gate
- **Automatically stored** in HiveMind for future reference

This is the ALAI AI Factory in action.

6. Quick Reference

Key commands, file paths, and configuration

6.1 Key Commands

Key Commands

Mission Control

```
node ~/system/tools/mc.js list                    # List open tasks
node ~/system/tools/mc.js show <id>               # Task details
node ~/system/tools/mc.js add "title" -p H        # Create high-priority task
node ~/system/tools/mc.js done <id> "reason"      # Complete task
node ~/system/tools/mc.js next-task               # Get next eligible task
```

Pi-Orchestrator

```
node ~/system/kernel/pi-orchestrator.js status    # Fleet health + worker pool
node ~/system/kernel/pi-orchestrator.js stop      # Stop daemon
launchctl kickstart -k gui/501/com.john.pi-orchestrator  # Restart daemon
```

Pipeline Engine

```
node ~/system/kernel/pipeline-engine.js status <task-id>    # Pipeline status
node ~/system/kernel/pipeline-engine.js advance <task-id>   # Advance to next stage
node ~/system/kernel/pipeline-engine.js create-pipeline <id> # Create pipeline for task
```

Virtual Companies

```
bash ~/system/tools/company.sh list               # List all companies
bash ~/system/tools/company.sh info CodeCraft      # Company details
```

Minions

```
node ~/system/tools/minion.js run "task" --project <path>   # Run minion
node ~/system/tools/minion.js status                         # Running minions
node ~/system/tools/minion.js list                           # Completed runs
```

HiveMind

```
node ~/system/agents/hivemind/hivemind.js query "question"  # Semantic search
node ~/system/agents/hivemind/hivemind.js post agent type "msg"  # Store knowledge
```

Key File Paths

Path	Purpose
`~/system/kernel/pi-orchestrator.js`	Main orchestrator daemon
`~/system/kernel/pipeline-engine.js`	Multi-company pipeline
`~/system/tools/mc.js`	Mission Control CLI
`~/system/tools/minion.js`	Minion executor
`~/system/agents/hivemind/`	HiveMind knowledge base
`~/system/config/pi-orchestrator-config.json`	Orchestrator config
`~/system/config/ollama-fleet.json`	Model fleet config
`~/companies/`	Virtual company definitions
`~/system/databases/mission-control.db`	MC task database
`~/system/databases/minions.db`	Minion run history
`~/system/logs/pi-orchestrator.log`	Orchestrator log

Agent System Improvements — Multi-Team Orchestration (2026-03-30)

Agent System Improvements — Multi-Team Orchestration (2026-03-30)

Source: IndyDevDan video "One Agent Is NOT ENOUGH: Agentic Coding BEYOND Claude Code"
Analysis: Plan agent + Devil's Advocate + Petter Graff (AI Architect) **Status:** IMPLEMENTED

Background

CEO reviewed IndyDevDan's multi-team agentic coding system (PI agent harness) and requested improvements to ALAI's virtual company/agent system. Video covers: three-tier architecture (Orchestrator → Leads → Workers), persistent mental models, domain write-locking, skills sharing, config-driven teams, multi-perspective consensus, and conversation logs.

Gap Analysis

What ALAI Already Does Better

- **14 specialized companies** vs 3 generic teams — more sophisticated domain pipeline
- **QA-19 quality gate** (19-point verification) — no equivalent in IndyDevDan's system
- **21 ZAKONs** (behavioral laws) with hook enforcement — codified governance
- **RAG/HiveMind infrastructure** (23K+ entries, Qdrant vector search) — shared knowledge base
- **Local AI tier routing** (Ollama ANVIL + FORGE) — multi-backend model routing
- **skill-improver.js** — skills improve from failures (NM i KI 2026 pattern)

Gaps Identified

Gap	Description	Impact
Lead delegation not enforced	Lead agents could write code (behavioral rule only)	HIGH
Write-lock not enforced	YAML allowed_paths existed but hooks didn't check them	HIGH
No session conversation logs	Agents couldn't see what other agents did	MEDIUM-HIGH
RAG-first was advisory only	Agents skipped RAG queries with no consequence	HIGH
HiveMind posts unstructured	Free-form text, no company/domain/pattern tags	MEDIUM

Rejected Proposal: Per-Agent Mental Models

Architect recommendation: Do NOT implement persistent expertise.md files per agent. Reason: creates 4th knowledge layer (HiveMind + Knowledge DB + RAG cache + expertise.md) that will diverge. Instead, enforce RAG-first as blocking and improve HiveMind post quality with structured tags. This gives 80% of the benefit with zero new infrastructure.

Implemented Changes

Faza 1: Lead YAML Constraints + Write-Lock Enforcement

Lead YAML updates (16/16 companies): All lead.yaml files now have:

```
constraints:
  allowed_paths:
    - "~/companies/<CompanyName>/**"
  forbidden_paths:
    - "~/projects/**"
    - "~/ALAI/products/**"
    - "~/system/**"
    - "~/.claude/**"
  write_locked: true
```

Leads can READ everything but only WRITE to their own company directory.

Write-Lock Script: `~/system/tools/agent-write-lock.py`

- Called by hook system on Write/Edit operations
- Reads agent identity from `/tmp/builder-session-active`
- Looks up agent's YAML constraints
- Blocks writes outside allowed_paths
- Pure stdlib Python (no PyYAML dependency)

Faza 2: Session Conversation Logs

Tool: `~/system/tools/session-workspace.sh` Commands:

- `create <mc-task-id>` — Creates shared session directory
- `report <mc-task-id> <agent-type> <company>` — Writes structured JSON report (stdin)
- `read <mc-task-id>` — Shows all agent reports as markdown
- `clean [--older-than 24h]` — Cleanup

Agent Integration:

- `builder.md` — Step 3: Read session reports on boot. Step 4c: Write report on completion.
- `validator.md` — Session Awareness section: Read builder reports before validation.

Report schema: agent_type, company, task_summary, files_written, key_decisions, blockers, verification.

Faza 3: RAG-First Blocking + Structured HiveMind Tags

RAG Enforcement Upgrade:

- `~/.claude/hooks/lib/rag_first_enforcer.py` upgraded with path classification
- `~/projects/**` and `~/ALAI/products/**` → **BLOCKING** (exit 2 = Write/Edit denied)
- `~/system/**` → advisory (warn only)
- `~/companies/**` → always allowed
- Config: `~/.claude/hooks/config/rag-enforcement.json`

Structured HiveMind Posts:

- `~/system/tools/hivemind-post-structured.sh`
- Format: `[TYPE] [Company] [domain] [Project] pattern: message`
- Data includes: company, domain, pattern, project as JSON metadata
- `builder.md` Step 5 updated to use structured format

Architecture Diagram

```
CEO (Alem)
  └── John (Orchestrator, Opus)
        ├── company.sh → Route to company
        ├── consensus-query.js → Multi-company consensus (planned)
        │
        ├── CodeCraft (Lead: delegate-only, write-locked)
        │     ├── Builder (Sonnet, allowed_paths enforced)
        │     └── Validator (Sonnet, read-only)
        │
        ├── Proveo (Lead: delegate-only, write-locked)
        │     ├── QA Builder
        │     └── Audit Validator
        │
        └── ... (14 companies total)

Shared Infrastructure:
  ├── HiveMind (23K entries, structured tags)
  ├── RAG-first hook (BLOCKING for project files)
  ├── Session workspace (/tmp/session-{task-id}/)
  └── Write-lock enforcement (agent-write-lock.py)
```

Files Changed

File	Change
~/companies/*/agents/lead.yaml (x16)	Added constraints, write_locked: true
~/system/tools/agent-write-lock.py	NEW — write-lock enforcement script
~/system/tools/session-workspace.sh	NEW — shared session log tool
~/system/tools/hivemind-post-structured.sh	NEW — structured HiveMind post helper
~/.claude/hooks/lib/rag_first_enforcer.py	Upgraded — blocking for project paths
~/.claude/hooks/config/rag-enforcement.json	Updated — project_mode: blocking
~/.claude/agents/builder.md	Updated — session awareness + structured posts
~/.claude/agents/validator.md	Updated — session awareness

Future Work

- **Consensus Query Tool** — Build when concrete architectural decision needs multi-company synthesis
- **Per-company cost tracking** — Token attribution per company per task
- **Lead operationalization** — Lead YAMLs are structural stubs; define real orchestration logic when needed