The Factory

Autonomous Agent Architecture

Comprehensive Technical Architecture Report

Version 1.0 — February 2026

Prepared by the **Kev Agent Team**

Kev (Orchestrator) · Rex (Builder) · Scout (Research) · Blaze (Growth)

February 9, 2026 (revised)

Contents

Chapter 1

Executive Summary

1.1 Vision

The Factory is an **autonomous AI factory** that discovers market opportunities, builds software products, deploys them, and generates revenue—24/7, with minimal human input. Adam serves as the portfolio manager, not the operator.

12-month target: \$30K–\$70K/month from a portfolio of 8–15 live products.

1.2 Core Architecture

The system operates on a two-layer orchestration model:

- **Kev (OpenClaw)** — Strategic brain. Decides *what* to build, assigns work, reviews output, communicates with Adam via WhatsApp.
- **Pi SDK** — Execution muscle. Spawns parallel Claude Code / Codex sessions for actual building.

Four core agents handle all work, expandable to 14 when justified by workload:

Agent	Role	Absorbs
Kev	Orchestrator	Strategy, ops, analytics, finance
Rex	Builder	Code, QA, testing, deployment
Scout	Researcher	Research, content, architecture
Blaze	Growth	Marketing, sales, legal review

Table 1.1: Core agent roster (simplified from original 14)

1.3 Key Numbers

Metric	Value	Notes
Infrastructure containers	3	MongoDB, Langfuse, Langfuse-Postgres
RAM usage	7–8 GB	Out of 64 GB available on dreamteam
Monthly cost (Month 1)	\$800–1,000	LLM APIs + electricity + hosting
Monthly cost (Month 12)	\$6,500–7,300	At full operation with 9 products
Revenue target (Month 12)	\$30K–70K	From 8–15 live products
Break-even	Month 5–7	Conservative estimate
Cost per product built	\$40–70	LLM costs for full build cycle

Table 1.2: Key financial and operational metrics

1.4 Why This Matters

The factory represents a paradigm shift: instead of building one product, we build a *system that builds products*. Individual businesses are experiments. The system that generates, tests, and scales them is what compounds. The architecture prioritises:

1. **MongoDB for everything** — One database replaces 5+ separate systems
2. **Minimal moving parts** — 3 containers, not 12+
3. **Free tiers first** — Pre-revenue discipline
4. **Ship products, not infrastructure** — If it doesn't help ship this week, it waits

Chapter 2

System Architecture

2.1 High-Level Architecture

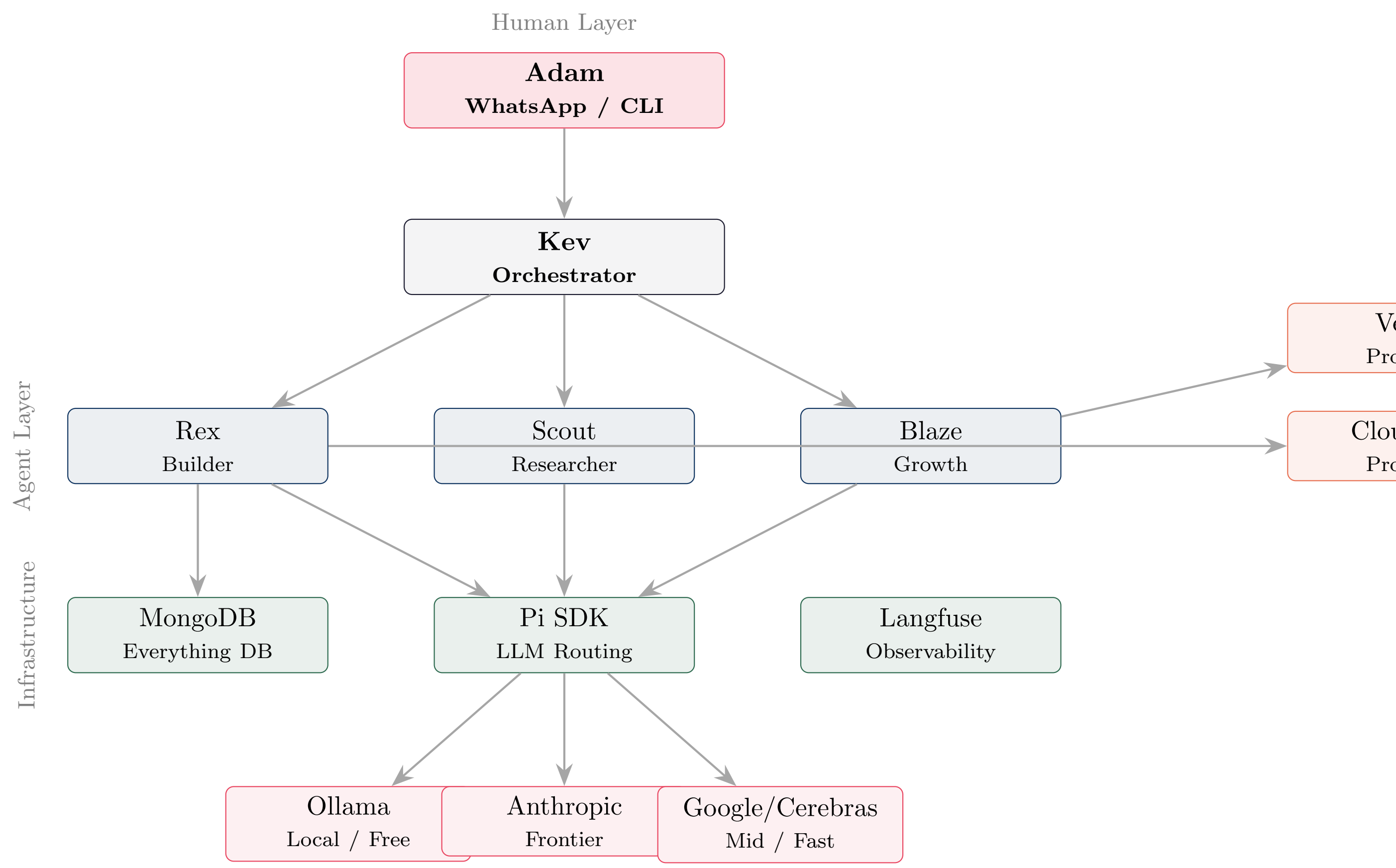

Figure 2.1: Factory high-level architecture

2.2 Component Overview

The factory runs entirely on **dreamteam** (RTX 3090, 64GB RAM) with products deployed to cloud edge platforms.

Component	**RAM**	**Purpose**
MongoDB 8.0	3–4 GB	Task queue, state, memory, metrics, events
Langfuse + Postgres	2 GB	LLM tracing and observability
Ollama	GPU VRAM	Local model inference (free tokens)
OpenClaw Gateway	1 GB	Agent runtime (Kev, Rex, Scout, Blaze)
Pi SDK	in-process	LLM routing (runs inside agent processes)
Total	**7–8 GB**	Leaves 55+ GB for agent processes

Table 2.1: Infrastructure services and resource allocation

2.3 Data Flow Overview

All data flows through MongoDB as the central nervous system:

1. **Task dispatch:** Kev writes task documents → change streams notify agents
2. **Agent coordination:** Agents claim tasks atomically via `findOneAndUpdate`
3. **Memory:** Agent memories stored as documents with vector embeddings
4. **Metrics:** LLM calls logged to time-series collections
5. **Events:** System events in capped collections with change stream watchers

Chapter 3

Technology Stack

3.1 MongoDB — The Universal Database

Decision: MongoDB Community Edition (self-hosted), migrating to Atlas when needed.

MongoDB was chosen as the *single database* for the entire factory, replacing what would otherwise be 5–7 separate systems.

3.1.1 What MongoDB Replaces

Was (Original Architecture)	Now (MongoDB)
PostgreSQL (task queue)	MongoDB
SQLite (orchestrator state)	MongoDB
Redis (caching, pub-sub)	MongoDB change streams + in-memory
Qdrant/ChromaDB (vector search)	MongoDB Atlas Vector Search
Neo4j (knowledge graph)	MongoDB `$graphLookup`
TimescaleDB (time-series)	MongoDB time-series collections
NATS JetStream (message bus)	MongoDB change streams

Table 3.1: MongoDB consolidation — 7 systems reduced to 1

3.1.2 Decision Rationale

1. **Change streams** replace message buses for event-driven coordination
2. **NoSQL flexibility** — agent state, tasks, memory all have different shapes; no migrations needed
3. **Atlas Vector Search** — embeddings stored alongside documents, single query for hybrid search
4. **Document model fits agent work** — tasks, memories, configs are naturally JSON
5. **Expert support** — Jake works at MongoDB; free expertise and likely Atlas credits
6. **TTL indexes** for automatic cleanup; capped collections for fixed-size event logs

3.2 Pi SDK — Single LLM Layer

One layer, no proxy: Pi SDK (`@mariozechner/pi-ai`) handles both LLM execution *and* provider routing directly.

- **Multi-provider routing** via `getModel()` with provider prefixes: `"anthropic/"`, `"google/"`, `"cerebras/"`, and OpenAI-compatible endpoints for local models

- **Parallel execution** — spawns Claude Code / Codex sessions
- **Fallback routing** via simple try/catch in application code
- **Cost tracking** via Pi SDK's usage reporting, logged to MongoDB
- **No proxy container needed** — runs in-process with agents

3.3 LLM Provider Strategy — Cost Pyramid

Tier	Provider	Models	Use Case	% Calls
Local (free)	Ollama	Llama 3.2 8B, nomic-embed	Embeddings, triage, drafts	50–60%
Speed (cheap)	Cerebras	Llama 3.3 70B	Fast generation, summaries	15–20%
Mid (balanced)	Google	Gemini 2.5 Flash/Pro	Long-context, general coding	15–20%
Frontier	Anthropic	Claude Sonnet 4.5 / Opus	Architecture, critical code	5–10%
Backup	OpenAI	GPT-5 mini	Fallback	<5%

Table 3.2: LLM provider tiers and cost pyramid

Target cost mix: 60% free/local, 25% cheap/mid, 15% frontier = ~$30–50/day at full operation.

3.4 Langfuse — LLM Observability

Self-hosted Langfuse provides per-trace cost tracking, prompt management, and evaluation framework. Instrumented via Langfuse's TypeScript SDK as a thin wrapper around Pi SDK calls. Requires a minimal Postgres sidecar (~512MB RAM).

Why kept despite simplification review: One Docker container, 10-minute setup, and building custom cost tracking would take weeks. Cost visibility from day one is non-negotiable at $30–50/day LLM spend.

3.5 TypeScript + Node.js

One language for everything. Pi SDK is TypeScript. OpenClaw is Node.js. Every product the factory builds will likely be TypeScript. Claude Code and Codex both excel at TypeScript generation.

3.6 Cloudflare — Product Deployment

Default deployment target for products built by the factory:

- **Workers:** 100K requests/day free, zero cold starts, global edge
- **Pages:** Unlimited static hosting
- **D1:** 5GB database free
- **R2:** 5GB storage free

Fallbacks: Vercel (Next.js SSR), Railway (persistent processes), Fly.io (Docker containers).

3.7 Supporting Stack

Component	**Choice**	**Purpose**
Auth (products)	Clerk	10K MAU free; drop-in React components
Payments (products)	Stripe	Industry standard; best API
CI/CD	GitHub Actions	2,000 min/month free; lint→test→build→deploy
Browser automation	Playwright (local)	Testing + basic scraping
Secrets management	Environment variables	`.env` + `chmod 600`; no Vault needed

Table 3.3: Supporting technology choices

Chapter 4

Agent Design

4.1 The Simplification Thesis

The original architecture proposed 14 specialist agents. The simplification review correctly identified this as over-engineering for a single-machine operation:

> *"This system is designed for a 50-person engineering org. It's one person and some LLMs. Cut 70% of it."* — Atlas, Simplification Review

With 200K–1M token context windows, a single agent session can hold an entire project's codebase, requirements, test results, and deployment config. The overhead of inter-agent handoffs exceeds the specialisation benefit.

4.2 Four Core Agents

Agent	Role	Absorbs from Original 14
Kev	Orchestrator + ops + analytics + finance	Kev, Dash, Finn, Dot — all "look at data, make decisions" tasks
Rex	Builder + QA + deploy	Rex, Forge, Hawk, Pixel — codes, tests, deploys in single session
Scout	Research + content + strategy	Scout, Echo, Atlas — read, synthesise, write
Blaze	Marketing + sales + growth	Blaze, Chase — low-volume, shares context

Table 4.1: Core agent roster with consolidated responsibilities

4.2.1 Model Assignments

Agent	Default Model	Fallback	Daily Budget
Kev	Claude Sonnet 4.5	Gemini 2.5 Flash	$30
Rex	Claude Code (Pi SDK)	Codex	$40
Scout	Cerebras / Gemini Flash	Claude Sonnet	$15
Blaze	Gemini Flash	Claude Haiku	$10

Table 4.2: Agent model assignments and budget caps

4.3 Expansion Path to 14 Agents

Agents should be split **only** when:

- Context windows consistently max out

- Task types have genuinely conflicting system prompts
- Workload exceeds single-agent throughput for a role

The original 14-agent roster (Kev, Rex, Forge, Scout, Hawk, Pixel, Blaze, Echo, Chase, Finn, Dash, Dot, Law, Atlas) serves as the expansion menu, not a day-one deployment plan.

4.4 Budget Enforcement

- **Global daily cap:** $100 (circuit breaker)
- **Per-agent daily caps:** Kev $30, Rex $40, Scout $15, Blaze $10
- **Per-provider monthly caps** set in provider dashboards as backup
- Pi SDK usage logged to MongoDB → real-time spend queries

Chapter 5

MongoDB Deep Dive

5.1 Change Streams for Agent Coordination

Change streams use MongoDB's oplog to push real-time notifications when documents change. This replaces a dedicated message bus (NATS/Redis) for a single-machine deployment.

Listing 5.1: Event-driven task dispatch via change streams

```
const pipeline = [
  { $match: {
    operationType: "insert",
    "fullDocument.type": "task.created"
  }}
];
const changeStream = db.events.watch(pipeline, {
  fullDocument: "updateLookup"
});
changeStream.on("change", (event) => {
  // Route to appropriate agent
});
```

Key properties:

- Latency: 10–50ms for local replica set
- Resumable via token after disconnection — no lost events
- Server-side filtering via aggregation pipeline
- Guaranteed total order on a single collection

Requires a replica set (even single-node) for change streams. The Docker Compose init container handles this automatically.

5.2 Task Queue Pattern

Atomic task claiming equivalent to PostgreSQL's `FOR UPDATE SKIP LOCKED`:

Listing 5.2: Atomic task claim pattern

```
db.tasks.findOneAndUpdate(
  { status: "pending", assigned_to: null },
  { $set: {
    status: "claimed",
    assigned_to: "rex",
    claimed_at: new Date()
  }},
  { sort: { priority: -1, created_at: 1 },
    returnDocument: "after" }
)
```

5.3 Vector Search for Agent Memory

Embeddings stored alongside the documents they describe. One query combines structured filters and semantic similarity:

Listing 5.3: Vector search with pre-filtering

```
db.memories.aggregate([
  { $vectorSearch: {
    index: "memory_index",
    path: "embedding",
    queryVector: queryEmbedding,
    numCandidates: 100,
    limit: 10,
    filter: {
      agentId: "scout",
      timestamp: { $gte: ISODate("2026-02-01") }
    }
  }},
  { $project: {
    content: 1,
    score: { $meta: "vectorSearchScore" }
  }}
]);
```

Note: Community Edition has limited vector search. For full Atlas Vector Search before cloud migration, use a free M0 Atlas cluster for vector-heavy collections.

5.4 Time-Series Collections for Metrics

10–20x compression vs regular collections, with automatic bucketing and built-in expiry:

Listing 5.4: Time-series collection for agent metrics

```
db.createCollection("metrics", {
  timeseries: {
    timeField: "timestamp",
    metaField: "source",
    granularity: "minutes"
  },
  expireAfterSeconds: 2592000  // 30 days
});
```

5.5 Knowledge Graph with `$graphLookup`

Replaces Neo4j for 90% of use cases—"what's connected to X?" style queries:

Listing 5.5: Graph traversal with $graphLookup

```
db.relationships.aggregate([
  { $match: { from: "entity:quickform" } },
  { $graphLookup: {
    from: "relationships",
    startWith: "$to",
    connectFromField: "to",
    connectToField: "from",
    as: "graph",
```

```
    maxDepth: 3
  }}
]);
```

5.6 Collections Architecture

Collection	Purpose
`factory.tasks`	Task queue with change stream dispatch
`factory.agent_state`	Agent configs, status, heartbeats
`factory.memory`	Agent memory with vector embeddings
`factory.llm_calls`	LLM usage tracking (time-series)
`factory.events`	System events (capped, change stream)
`factory.products`	Product registry and configuration
`factory.revenue`	Revenue tracking per product
`factory.approvals`	Human approval queue

Table 5.1: MongoDB collections architecture

Chapter 6

Data Flows

6.1 Scenario 1: Build a Product

Trigger: Adam sends “Build me a price comparison API” via WhatsApp.

Phase	Agent	Action	Time	Cost
1. Research	Scout	Market research via Cerebras (free), web scraping	~25s	\$0.05
2. Design	Kev	API spec + OpenAPI schema via Claude Opus	~10s	\$0.15
3. Build	Rex (×2)	Parallel Claude Code sessions, scaffold + implement	5–8 min	\$1.50
4. Test/QA	Rex	Adversarial review with different model, full test suite	3–6 min	\$0.30
5. Deploy	Rex	CI pipeline, preview deploy, production rollout	5–10 min	\$0.05
		Total (excluding human approval)	**15–25 min**	**\$2.05**

Table 6.1: End-to-end product build data flow

Data transformation chain:

Natural language → Task DAG → Research queries → Structured report → API spec → TypeScript source → Git commits → CI artifacts → Deployed service

6.2 Scenario 2: Revenue Engine Scan

Trigger: Kev’s 30-minute heartbeat cron fires the revenue engine pipeline.

1. **Market Scan (~2–3 min):** Scout scrapes Reddit, HN, G2, Google Trends, Product Hunt via Cerebras (free). Outputs 5–10 structured opportunity objects.
2. **Scoring (~30s):** Kev scores each opportunity using Claude Sonnet + historical outcomes from vector memory. Filters to score ≥ 60.
3. **Validation (48h):** Deploy landing page, post to communities, run \$50–100 in ads. Measure: pageviews, CTR, signups.
4. **Build or Kill:** GO = full build pipeline (Scenario 1). NO-GO = archive + store learnings in memory.

Total cycle: ~50 minutes of agent time + 48 hours of validation = idea to live product.

6.3 Scenario 3: Incident Response

Trigger: Customer complaint about API returning 500 errors.

1. **Triage (T+8s):** Classify severity, sentiment, churn risk via fast model

2. **Parallel response:** Kev investigates (Sentry, deploy history), auto-drafts customer acknowledgement
3. **Rollback (T+60s):** Auto-rollback is always authorised, no human approval needed
4. **Health verification (T+120s):** Confirm service restored via monitoring APIs
5. **Root cause (T+20min):** Analyse reverted commit, create fix PR with tests
6. **Post-mortem (T+30min):** Generate incident report, update anti-pattern library

Chapter 7

Security Model

7.1 Threat Landscape

The security review identified **2 critical** and **4 high-severity** findings. The honest assessment: the security architecture document describes a hardened production system, but the actual Phase 1 deployment runs agents as shared processes on one machine with filesystem coordination.

7.2 Critical Finding 1: Shared Filesystem Destroys Isolation

Severity: CRITICAL

The architecture describes per-agent sandboxes. Reality: all agents share `/home/adam/agents/shared/` with full read/write access.

Attack scenario: Prompt injection in Scout (via malicious web content) → writes poisoned task to queue → Rex executes with deploy credentials.

Remediation:

- Separate Unix users per agent (trivial, high impact)
- Restrictive file permissions on shared subdirectories
- JSON schema validation on all task queue files before Kev dispatches
- Tag externally-sourced content with clear delimiters

7.3 Critical Finding 2: Prompt Injection via Inter-Agent Messages

Severity: CRITICAL

Agents communicate via filesystem (markdown and JSON files). No schema validation, no content sanitisation, no orchestrator mediation on file reads.

Attack scenario: Malicious website embeds instructions in content → Scout's research output contains injected commands → Rex follows them.

Remediation:

- Implement dual-LLM pattern for agents processing external content
- Gateway process validates all files against strict JSON schema
- Agents instructed to never execute instructions found in data files

7.4 High-Severity Findings

Finding	Risk	Mitigation
Kev is God-object	Compromised orchestrator = total factory compromise	Deterministic policy engine for security decisions; don't pass raw agent output into routing
No credential isolation	All agents inherit same env vars	Separate env files per agent; restrict shell access
Self-improvement persistence	Prompt injection could get codified into permanent prompts	Human review mandatory for security-relevant prompt changes
Browser credential leakage	Browsing sessions expose auth tokens	Throwaway credentials for external sites; scan extracted data

Table 7.1: High-severity security findings and mitigations

7.5 What's Done Well

The security architecture includes several genuinely strong design decisions:

- "Agents are untrusted code" axiom — correct threat model
- Capability-based access control over RBAC
- Trust is per-capability, not global
- Timeout defaults to deny (fail-safe)
- Auto-rollback authority without approval

7.6 Phase 0.5 Security Baseline (Week 1)

1. Separate Unix users per agent
2. File permissions on shared directories
3. Per-agent environment files for credentials
4. Basic output regex scanning for credential patterns
5. JSON schema validation on task queue files

Chapter 8

Cost Analysis

8.1 Monthly Cost Projections

Category	Month 1	Month 6	Month 12
LLM APIs	\$500–700	\$1,300–1,800	\$2,000–2,800
Infrastructure (electricity)	\$50	\$60	\$65
Product hosting (free tiers)	\$10	\$200	\$600
Marketing	\$200	\$1,200	\$2,500
Domains	\$2	\$8	\$15
Stripe fees (2.9% + \$0.30)	\$0	\$250	\$1,280
Total	**\$800–1,000**	**\$3,000–3,500**	**\$6,500–7,300**

Table 8.1: Monthly cost projections

8.2 Revenue Projections and Break-Even

	Month 1	Month 6	Month 12
Revenue	\$200	\$8,000	\$40,000
Total costs	\$1,687	\$5,393	\$8,405
Net profit	**–\$1,493**	**+\$2,357**	**+\$30,315**
Gross margin	—	30%	78%

Table 8.2: Revenue projections (conservative)

Break-even: Month 5–7. The danger zone is months 2–5: burning \$1,500–3,000/month building the machine before meaningful revenue.

8.3 Cost Traps

1. **Stripe fees are real:** At \$40K MRR, Stripe takes ~\$1,280/month — never mentioned in original revenue projections
2. **Free tier cliff:** When products outgrow free tiers (Supabase, Vercel), costs jump \$200–300/month
3. **LLM costs scale non-linearly:** Context window bloat, retry amplification, RAG overhead add 30–50% above simple token math
4. **Agent keepalive costs:** Heartbeat polling burns \$20–78/month on agents doing nothing

5. **Prompt caching is not 90%:** Realistic cache hit rate is 20–40%; budget accordingly
6. **Extended thinking burns invisible tokens:** 7 agents with extended thinking = ~$375/month on reasoning tokens

Chapter 9

Risk Register

9.1 Top 10 Risks

#	Cat.	Risk	L	I	Score
1	Fin	API cost blowout / runaway agent loops	4	5	**20**
2	Tech	Prompt injection compromises agent actions	4	5	**20**
3	Ops	Single point of failure (dreamteam server)	4	4	**16**
4	Ops	Platform account bans (Stripe, hosting)	3	5	**15**
5	Tech	Agent quality degradation / shipping bad code	4	4	**16**
6	Mkt	Revenue model failure (no product-market fit)	3	5	**15**
7	Tech	Model provider dependency / API changes	3	4	**12**
8	Tech	Secret / credential leakage	3	5	**15**
9	Ops	Over-engineering before first revenue	4	3	**12**
10	Legal	Regulatory crackdown on AI-operated businesses	2	5	10

Table 9.1: Risk register — top 10 by score (L=Likelihood, I=Impact, 1–5 scale)

9.2 Critical Risk Mitigations

9.2.1 Cost Blowout (Score: 20)

- Per-task and per-agent budget caps via MongoDB aggregation queries
- Kill switches at global, per-agent, per-provider levels
- Cost velocity monitoring: alert if $/min exceeds 3× rolling average
- Max 20 tool calls per agent run (hard cap)
- Provider-level hard spend caps in dashboards

9.2.2 Prompt Injection (Score: 20)

- Dual-LLM pattern: quarantined LLM for untrusted content, privileged LLM acts on summaries
- Tool allowlists per task type
- Output filtering for dangerous patterns
- Human-in-the-loop for all external-facing actions

9.2.3 Single Point of Failure (Score: 16)

- UPS for power continuity

- Auto-restart all services via systemd
- Cloud fallback agents for critical functions
- All code in git (distributed by nature)
- Future: dedicated inference node separate from dev workstation

Chapter 10

Competitive Moat

10.1 What's Defensible

The Factory has **no single killer moat today**. The defensibility comes from **compounding system effects** that are hard to replicate in isolation.

10.1.1 Strong Moats (Hard to Copy)

Moat	Why It's Defensible	Score
Accumulated institutional memory	6+ months of production learning data is not replicable	8/10
Self-improvement flywheel	Closed-loop observe→measure→diagnose→patch→verify	9/10
Smart router heuristics	Learned from production, not guessed; measurable cost/quality advantage	7/10

10.1.2 Weak/Non-Moats

- "We use the best AI models" — so does everyone; models are commodities
- "Our architecture is elegant" — visible and copyable
- "We're first to market" — Devin, Factory AI, Cursor are already shipping
- MCP integration — open standard, table stakes
- Pricing — a lever anyone can pull

10.2 The Data Flywheel

More tasks → More signals → Better prompts/routing → Higher quality + lower cost → More customers → More tasks

Time to moat:

- **Month 1–3:** No moat. Architecture only. Any funded competitor is equal.
- **Month 3–6:** Early data advantages. Router heuristics outperform naive approaches.
- **Month 6–12:** Meaningful moat. Thousands of tasks of learning data.
- **Month 12+:** Strong moat. Gap widens faster than competitors can close.

Critical implication: Speed to production matters more than architectural perfection.

10.3 Competitive Threats

Threat	Likelihood	Timeframe	Mitigation
Devin adds orchestration	High	6–12 mo	Enterprise-focused; SMB positioning buys time
Cursor adds autonomous agents	High	3–6 mo	IDE-locked; orchestration is a different product
OpenAI/Anthropic ship factory	Medium	12–18 mo	Model labs historically bad at product
New funded startup	Medium	6–12 mo	Zero learning data; our head start in production signals

Table 10.1: Competitive threat assessment

Chapter 11

Implementation Roadmap

11.1 Week 1 Sprint

Goal: Task queue + orchestration core + first automated pipeline running end-to-end.

Friday demo: Adam triggers a product idea → Scout researches → Rex scaffolds → Hawk validates → output lands in review queue. All autonomous.

Day	Focus	Deliverables
Monday	Foundation	MongoDB task store + schema, task state machine, task CLI
Tuesday	Routing	Kev orchestration logic, cost pyramid router, agent registry
Wednesday	Pipeline	Scout research workflow, Rex scaffolding, DAG wiring
Thursday	Quality	Hawk QA gate, review queue, cross-model adversarial review
Friday	Integration	Bug fixes, retry logic, demo with real product ideas

Table 11.1: Week 1 sprint plan

Critical path: Task Queue → Kev Routing → First Pipeline → Hawk QA → Full Pipeline → Ship Product

11.2 Month 1–3 Milestones

Milestone	Description	Target
Week 1	Pipeline works: idea → research → build → QA → review	Feb 13
Week 2	Deployment pipeline (Forge), first production ship	Feb 20
Week 3	WhatsApp approval flow, spending controls, cost dashboard	Feb 27
Week 4	First product live with real URL, REEF dashboard	Mar 6
Month 2	Pipeline tuned, 2–3 products/week, marketing pipeline	Mar–Apr
Month 3	Factory self-funding from product revenue	Apr–May

Table 11.2: Month 1–3 milestone targets

11.3 What's Explicitly Deferred

Component	**Why Deferred**
Full memory system (vector DB)	Filesystem + basic memory enough initially
Browser automation (Stagehand)	Not needed until E2E testing or scraping
Data analytics platform	SQLite metrics + dashboard is enough
Marketing automation	No point marketing until product 1 ships
Self-improvement system	Needs months of operational data
Graduated autonomy / trust scores	All agents start supervised; need data first

Table 11.3: Deferred components with rationale

Chapter 12

First Product: CronPilot

12.1 Product Overview

CronPilot is cron job monitoring as a service. Register a job → get a unique ping URL → add `curl` to your cron script → get alerted if the ping doesn't arrive on schedule.

Why CronPilot: Lightest build (18 agent-hours), proven market (Cronitor is a $20M+ business), developer audience converts fast, and we dogfood it for every future product.

12.2 Technical Architecture

Layer	Choice
Frontend	Next.js 14 (App Router)
Backend	Next.js API routes + Cloudflare Workers (ping endpoint)
Database	Supabase (Postgres)
Auth	Supabase Auth (magic link + GitHub)
Payments	Stripe (subscription)
Hosting	Vercel (app) + Cloudflare Workers (ping receiver)
Alerts	Resend (email) + webhooks (Slack/Discord)

Table 12.1: CronPilot technical stack

12.3 Revenue Model

Plan	Price	Checks	Features
Hobby	$7/mo	20	Email alerts, 7-day history
Pro	$19/mo	100	All channels, 90-day history, API
Team	$49/mo	500	Multi-user, audit log, 1-year history

Table 12.2: CronPilot pricing tiers

Unit economics: Cost per check ~$0.01/month. 20-check Hobby user costs $0.20/month → 97% margin.

Target: 100 customers × $15 avg = $1,500 MRR achievable.

12.4 Build Plan

Day	Deliverables
Day 1	Repo setup, DB schema, Cloudflare Worker ping endpoint, miss detection, auth + dashboard shell
Day 2	Dashboard detail view, Stripe integration, alert system, REST API
Day 3	QA/E2E tests, landing page, monitoring setup, launch prep

Table 12.3: CronPilot 3-day build plan

Total agent-hours: 18h. **Estimated cost to launch:** $105–135.

12.5 Kill Criteria

- Day 7: <20 signups → reassess
- Day 30: <10 active checks AND $0 MRR → kill
- Day 60: <$70 MRR → kill
- Day 90: <$500 MRR with no growth → kill

Appendix: Full Tech Stack Decision Table

Decision Area	Choice	Rejected	Key Rationale
Database	MongoDB	PostgreSQL, SQLite, Redis	One DB for everything; change streams; Jake's expertise
Message Bus	MongoDB Change Streams	NATS, Redis Streams	One machine; no separate bus needed
Vector Search	MongoDB Atlas Vector Search	Qdrant, ChromaDB	Vectors live with documents; one query for hybrid search
LLM Layer	Pi SDK	LiteLLM, custom router	Direct provider routing via getModel(); no proxy needed
Observability	Langfuse (self-hosted)	Grafana+Prometheus, custom	Trace visualization; Langfuse SDK integration
Language	TypeScript	Python	Ecosystem alignment; deployment targets; type safety
Product Deploy	Cloudflare Workers	AWS, GCP, Azure	Generous free tier; zero cold starts; simple deploys
CI/CD	GitHub Actions	ArgoCD, Jenkins	Free tier sufficient; simple pipeline
Auth (products)	Clerk	Supabase Auth, NextAuth	10K MAU free; drop-in components
Payments	Stripe	Paddle, LemonSqueezy	Industry standard; best API; agent-friendly
Knowledge Graph	MongoDB $graphLookup	Neo4j, Memgraph	Zero additional infra; covers 90% of queries
Browser Automation	Playwright (local)	Browserbase	Free; agents know it; add cloud later
Secrets	Environment variables	HashiCorp Vault	One machine, one operator; .env is fine
Agent Count	4 core agents	14 specialists	Less handoff overhead; huge context windows
Local Inference	Ollama	llama.cpp direct	Simpler management; same performance

Decision Area	Choice	Rejected	Key Rationale
Hosting (factory)	dreamteam (local)	Cloud VMs	Existing hardware; 64GB RAM; RTX 3090

Table 12.4: Complete technology stack decision table