NOA — ExecutiveCommanderChiefAgent

Definition & Purpose

NOA (sometimes called the **ExecutiveCommanderChiefAgent**) is the top-level orchestrator of the **ark-os-noa** platform. It acts like a CEO for the agent ecosystem: it translates high-level business goals into concrete plans, delegates work to Board Agents and **MicroAgentStacks**, and ensures that every deliverable meets business, technical, and compliance requirements.

Framework

- **Inputs:** high-level goals, success criteria, budgets, SLAs, risk appetite and constraints. NOA normalises these into a **WorkPlan**. Each plan captures tasks, checkpoints, deadlines and deliverables.
- **Outputs:** action plans, stack assignments, acceptance tests and post-mortems. For each goal NOA produces a package of artefacts (e.g. zip file and compiled PDF).
- **Control loop:** Sense → Plan → Act → Verify → Report. NOA constantly senses progress and risks, replans when necessary, acts by spawning or destroying **MicroAgentStacks**, verifies outputs against acceptance criteria, and finally reports to the business owner.

Goals

1. **Disambiguate and decompose:** convert ambiguous goals into measurable objectives and step-by-step tasks.
2. **Resource allocation:** assign Board Agents and MicroAgentStacks based on domain expertise, constraints and availability.
3. **Policy enforcement:** apply safety, security and legal policies; ensure no Docker-in-Docker (**Capsule/Full-Illusion** pattern) and maintain audit logs.
4. **Model selection:** orchestrate **ModelSelectorAgents** to pick appropriate AI models for each task, balancing accuracy, latency and cost.
5. **Packaging & archiving:** guarantee that outputs are packaged into deliverable artefacts (zip + PDF) and stored internally.

Capabilities

- **Decomposition & scheduling:** build dependency graphs, schedule tasks across stacks and board seats, and respect deadlines.
- **Auto-retry & escalation:** detect failures or blockers and retry tasks with backoff; when automation fails, summarise context and ask for human input.
- **Observability:** generate unique run IDs, attach traces and metrics, and centralise logs for all stacks.
- **Safety & compliance:** enforce licensing, vulnerability thresholds and secret scanning. Use outer BuildKit and containerd with sidecars rather than nested containers to avoid

security risks【43537238352704†L1068-L1088】.

Objects & Definitions

- **WorkPlan:** a structured representation of a goal → tasks → checkpoints → deliverables → review gates.
- **Assignment:** mapping between Board Agents, MicroAgentStacks and tasks; includes SLAs and ownership.
- **Trace:** evidence of inputs, actions, tools, models and outputs for audit and reproducibility.

Lifecycle

1. **Intake & Normalise:** accept a business goal and convert it into a WorkPlan.
2. **Resource Match:** choose which Board Agents and stacks are needed and spin them up.
3. **Execution:** coordinate tasks across microservices; check progress with periodic checkpoints.
4. **Validation & Packaging:** verify results, run security and licence scans, and package deliverables.
5. **Report & Archive:** summarise results, produce a post-run report, archive artefacts with retention policies.

Tools & Resources

NOA can invoke various tools through subordinate agents, including: web research, code & data analysis, file search, and automations. It delegates model selection to ModelSelectorAgents and leverages microservices to execute tasks. It works with the internal data plane (OCI registry, MinIO, Postgres/pgvector, Supabase) to store and retrieve artefacts, always within the trust boundary.# Board Agents — Executive Team of ark-os-noa

Definition & Role

The **Board Agents** sit at the top of the **ark-os-noa** organisation just below NOA. They are analogous to an executive board in a company: each agent owns a domain (strategy, operations, finance, legal, security, partnerships, research) and has authority to commission **MicroAgentStacks** to execute work. By design they are *few in number* but *broad in scope*—their purpose is to translate NOA's vision into specific missions, ensure alignment with ElementArk/DeFlex's business model, and provide governance across all stacks and agents.

Roster & Responsibilities

- **Strategy/CTO Agent** – Sets technical direction: system architecture, Capsule (Full-Illusion) adoption, environment policies (no Docker-in-Docker), cohesion across services.
- **COO Agent** – Owns operational runbooks, SLAs, scheduling and change management. Coordinates delivery timelines and resource utilisation.

- **CFO/FinOps Agent** – Manages budgets and spend telemetry. Optimises cost across compute, storage and model usage.
- **Legal/Compliance Agent** – Ensures licence compliance, data governance, export controls and regulatory adherence. Maintains policy frameworks.
- **Security Agent** – Enforces secrets management, supply-chain security, SBOM attestation and vulnerability thresholds. Gatekeeper for risk.
- **Growth/Partnerships Agent** – Curates ingestion roadmaps for repos, APIs and CRMs; drives ecosystem strategy and partnership integrations.
- **Digest Agent (R&D)** – Sits on the board as the research arm. Its role is to *digest everything* (code, data, SaaS, models) and surface insights. See `digest_agent.md` for details.

Operating Rules

1. **Delegation:** Board Agents can spin up one or more **MicroAgentStacks** to accomplish tasks. Each stack has its own **CommandChiefAgent** orchestrating the details, leaving the Board Agent to focus on strategy and oversight.
2. **Specialisation:** When a task requires sophisticated model selection, a Board Agent requests a **ModelSelectorAgent** to choose the most appropriate AI model or tool. This ensures tasks are executed with the right balance of cost, latency and accuracy.
3. **Governance:** Board Agents enforce policies across stacks—licensing, vulnerability gates, security posture, and budget limits. They maintain decision logs and risk registers for audit.
4. **Parallelism:** Multiple stacks can run concurrently. Board Agents schedule tasks to maximise throughput while respecting resource constraints.

Capabilities

- **Multi-project scheduling:** assign and monitor numerous tasks across different domains and stacks; handle dependencies and deadlines.
- **Cross-repo initiatives:** coordinate wide-sweep digest operations (e.g., SBOM/security posture across all repos) by commissioning multiple stacks.
- **Program governance:** maintain an overarching view of risks, mitigations, budget spend, and deliverable quality.
- **Policy enforcement:** integrate security scanners, licence gates, and compliance checks into the workflow.

Tools & Signals

Board Agents interact with the system through:

- **Research & analysis tools:** for web search, code parsing and data exploration within the current year's context.
- **Change control & telemetry:** CI/CD gates, policy engines (e.g. OPA), vulnerability scanners and cost dashboards.
- **Observability feeds:** real-time traces, metrics and logs aggregated from

MicroAgentStacks and sidecars. These signals inform decisions on scaling up/down stacks or raising alerts.

Relationship to Other Components

- **NOA:** Board Agents receive missions from NOA and report status back. They provide domain expertise and enforce governance while letting NOA handle high-level planning and cross-domain coordination.
- **MicroAgentStacks:** Board Agents are the owners of stacks. They commission stacks to achieve defined objectives and decommission them when tasks complete. Each stack operates autonomously but reports progress to its Board Agent.
- **ModelSelectorAgents:** When tasks require AI model inference, Board Agents request a ModelSelector to choose among local or hosted models. The selection is recorded in the trace for audit.
- **Digest Agent:** The Digest Agent is part of the Board but behaves like an R&D lab—collecting raw information, synthesising knowledge graphs and summarising findings for the board to act on.

By keeping the Board Agents separate from execution details yet close enough to enforce policy, ark-os-noa achieves a balance between **strategic oversight** and **operational agility**. # ModelSelectorAgents — Choosing the Right Tool for the Job

Purpose

A **ModelSelectorAgent** specialises in selecting the best AI model or tool for a given task. In the context of ark-os-noa, tasks vary widely—from reasoning and planning, to code analysis, to data transformation. Selecting the wrong model can waste resources or compromise privacy. The ModelSelector provides an intelligent arbitration layer, helping Board Agents and **MicroAgentStacks** achieve high quality results while respecting cost, latency and privacy constraints.

Framework

- **Inputs:** Each call to a ModelSelector includes a task description, input size (e.g. document length, number of files), the privacy tier (public, sensitive, confidential), latency budget, and a cost cap. These parameters come from the requesting agent (often a Board Agent or CommandChiefAgent).
- **Decision Graph:** The ModelSelector applies a decision graph:
  1. **Task classification** – Is this reasoning/planning, bulk transformation, code/data manipulation, or something else?
  2. **Complexity estimation** – How large or intricate is the input? This influences whether to use a bigger model or a lightweight one.
  3. **Model/Tool selection** – Choose from a catalogue of available models (remote APIs, local models served via llama.cpp/Ollama, code runners, data converters) using heuristics or learned policies.
  4. **Guardrails assertion** – Check licensing, privacy levels and security requirements.

For example, confidential data must stay on-prem and use local models.

- **Outputs:** A plan specifying the chosen model or tool, the expected cost/latency, and a rationale. The rationale becomes part of the execution **Trace**, enabling auditing and future optimisation.

Default Policy

The default policy can be tuned, but common guidelines include:

1. **Reasoning / Planning tasks:** Use high-quality generalist models (e.g. GPT-5). These tasks benefit from advanced reasoning and tolerance for slower latency when results matter.
2. **Bulk transforms / formatting:** Use fast, cost-efficient models; they handle repetitive conversions without needing deep reasoning.
3. **Code & data tasks:** Prefer dedicated code analysis tools or local runtimes for safety. Use sandboxed execution to evaluate code or parse data. Employ smaller codex models when summarising code.
4. **Offline/local fallbacks:** If the privacy tier demands on-prem processing or if network latency is unacceptable, use local models served via llama.cpp, vLLM or similar frameworks. This reduces latency and eliminates external data exposure.

Tools & Telemetry

- **Model catalogues:** The selector maintains metadata about available models—accuracy, context limits, token costs, latency benchmarks, licensing and hardware requirements. It syncs with the local model server and remote provider APIs.
- **Cost/latency forecaster:** Predicts cost and latency using historical telemetry and dynamic system load. This helps decide when to use a cheaper but slower model vs. a more expensive high-performance one.
- **Performance feedback:** The selector ingests feedback after tasks complete (e.g. success, error rate, user satisfaction). Over time it learns to better match tasks to models.

Relationship to Other Components

- **Board Agents:** Request ModelSelector assistance when their tasks involve AI/ML. They set budgets and specify privacy tiers. The ModelSelector returns a plan and rationale.
- **MicroAgentStacks:** CommandChiefAgents invoke ModelSelectors inside their stacks when a task requires AI processing. This ensures each stack uses consistent policies and optimal models.
- **NOA:** Maintains overarching policies for model selection (allowed licences, vulnerability gates, GPU quotas). The ModelSelector enforces these policies and logs decisions back to NOA's audit trail.

Benefits

- **Efficiency:** Avoids blindly calling the largest or default model for every task, saving compute and cost.

- **Compliance:** Ensures tasks adhere to privacy and licensing requirements—confidential data stays internal.
- **Transparency:** Provides a clear rationale for each selection so decisions can be audited and improved.
- **Extensibility:** New models or tools can be added to the catalogue; the decision graph can be refined with new criteria or learned policies.

By delegating model/tool choice to a dedicated ModelSelectorAgent, ark-os-noa keeps business logic and AI expertise separate, resulting in better outcomes and traceable decisions. # MicroAgentStack — Cooperative Work Pods

Definition

A **MicroAgentStack** is a deployable cluster of cooperative agents assembled to accomplish a bounded objective. Think of it as a project team spun up on demand: each stack has its own **CommandChiefAgent** (the stack master), a set of specialised Operators, Adapters and Guards, and a dedicated workspace. Stacks can be created, scaled and destroyed rapidly, making them the primary execution units within ark-os-noa.

Composition

- **CommandChiefAgent (Stack Master):** Orchestrates the stack, decomposes tasks, assigns work to subordinate agents, monitors progress, resolves conflicts and enforces SLAs.
- **Operators:** Specialised agents that perform specific functions. Examples include code runners (execute code), data wranglers (transform data), doc generators (produce reports), testers (run unit/integration tests) and packagers (build zips, PDFs).
- **Adapters:** Connectors to external systems (repos, CRMs, APIs) and publishers to internal services (registry, MinIO, Postgres). Adapters abstract away details like auth and rate-limits.
- **Guards:** Policy enforcement points—security scanners, licence checkers, quality gates. They ensure the stack adheres to policies defined by NOA and the Board Agents.

Goals

1. **Deliver end-to-end outcomes:** A stack should own the entire life cycle of its objective—from cloning a repo to producing a digest report, from running tests to publishing a package.
2. **Scale horizontally:** Multiple stacks can be spun up concurrently when tasks are independent or parallelisable. This enables large scale operations like digesting hundreds of repos simultaneously.
3. **Clean teardown:** After completion, a stack cleans up its resources (containers, temporary volumes) and archives logs, SBOMs and artefacts with proper retention policies.

Lifecycle

1. **Bootstrap:** Given inputs (e.g. repo URL, CRM base URL, model list), the CommandChiefAgent creates a **WorkPlan**, prepares the environment and mounts necessary sidecars. It avoids Docker-in-Docker by using **Capsule** sidecars to talk to the outer BuildKit/containerd environment【43537238352704†L1068-L1088】.
2. **Execute:** The stack runs its Operators in parallel where possible. Retrying tasks with exponential backoff ensures resilience; failures trigger controlled retries or escalation to the Board Agent.
3. **Validate:** Once tasks finish, Guards run acceptance tests (e.g. unit tests, SBOM scans, licence checks) and produce human-readable summaries. If acceptance criteria fail, the stack either retries or fails the WorkPlan.
4. **Package:** On success, the stack assembles outputs into deliverables (zip file, compiled PDF, JSON indices). It updates internal registries (OCI images, Postgres metadata, vector DB) and publishes logs and traces.
5. **Archive:** The stack removes its runtime environment and persists all logs, SBOMs, run IDs, and checksums. Retention policies decide how long to keep each artefact.

One-liners & Conventions

- Stacks are named by timestamps or descriptive identifiers (e.g. `stack-20250822-103045`).
- They maintain their own directory structure (`in/`, `work/`, `out/`, `logs/`) for clarity and reproducibility.
- Each stack produces a unique run ID and attaches it to all outputs and logs for traceability.

Relationship to Other Components

- **Board Agents:** Create and oversee stacks. Each stack reports to its Board Agent. Board Agents can run multiple stacks in parallel.
- **ModelSelectorAgents:** When a stack requires AI processing, the CommandChiefAgent requests a ModelSelector to choose the appropriate model and logs the rationale.
- **Digest Agent:** Often uses MicroAgentStacks to perform large-scale digestions across many repos or datasets. Each stack digests one or more sources and returns results to the Digest Agent.

MicroAgentStacks bring structure, scalability and reliability to ark-os-noa's execution model. By isolating work into bounded pods, the system can handle complex, parallel workflows without turning into a monolith. # Digest Agent — R&D Engine for ark-os-noa

Role & Position

The **Digest Agent** operates as the research and development arm of the Board Agents. Its primary mission is to "*digest everything*"—code repositories, datasets, documents, APIs, SaaS systems (including live CRMs) and even AI models. By analysing these sources, the Digest

Agent extracts structured knowledge, builds semantic indices, and surfaces insights that inform strategic decisions. Though part of the Board, it behaves like a self-contained lab, spinning up **MicroAgentStacks** to perform large-scale digestions.

Pipeline

1. **Discover:** Identify sources to digest. This includes scanning internal GitHub repos, listing connected APIs/CRMs, and reading the current model ingestion list. Discovery may rely on board directives or scheduled tasks.
2. **Fetch:** Clone or synchronise the source material. For code repos, perform a shallow clone and gather dependency lock files. For CRMs or APIs, pull metadata and sample records while respecting rate limits. Handle authentication using secure tokens from the secrets manager.
3. **Parse:** Use language-specific parsers (Python AST, ts-morph for JS/TS, go/ast, Rust syn, JavaParser) to analyse code and extract modules, functions, classes and call graphs. For API schemas, parse OpenAPI/GraphQL definitions. Build an **SBOM** to capture all packages and versions.
4. **Analyze:** Generate embeddings for code, documentation and data using models selected via the **ModelSelectorAgent**. Build a **knowledge graph** linking functions, data structures, APIs and entities. Identify external API calls, config surfaces and extension points. Apply entity linking to unify references across sources.
5. **Summarize:** Produce layered summaries: per file, per module, per repository and across repositories. Summaries highlight the system's purpose, architecture, dependencies, risks and extension points. The Digest Agent uses LLMs to craft human-readable reports and cross-links to original sources.
6. **Surface:** Publish outputs as markdown dossiers, dashboards and vector DB upserts. Persist `profile.json`, `system_card.md`, `kg.json`, and embeddings. Offer search and retrieval APIs for downstream agents.
7. **Secure:** Scan for secrets and vulnerabilities using tools like Trivy, Grype and Gitleaks. Classify findings by severity and quarantine sensitive information. Tag licences and export-control flags【43537238352704†L1068-L1088】.

Tools

- **Web research:** limited to current-year sources, retrieving official documentation and examples.
- **Language parsers & AST tools:** Python's `ast`, TS's `ts-morph`, Go's `go/ast`, Rust's `syn`, Java's `JavaParser`.
- **Security scanners:** Syft to produce SBOMs; Grype and Trivy to scan for vulnerabilities; Gitleaks to detect secrets; Semgrep for static analysis.
- **Embeddings & vector DB:** Sentence transformers or llama.cpp embedding models; pgvector or Qdrant to store vectors and link them to original files.
- **Visualization & reports:** Graph builders, markdown generators and PDF compilers.

Outputs

The Digest Agent delivers:

- **Digest reports:** Markdown documents (e.g. `2025-08-22_digest_report.md`) summarising findings.
- **Structured indices:** JSONL files representing the knowledge graph, call graph and embedding metadata. These feed search and retrieval APIs.
- **SBOM & security reports:** Comprehensive lists of dependencies and vulnerabilities.
- **Vector store entries:** Embeddings upserted to the chosen vector DB for semantic search.

Relationship to Other Components

- **Board Agents:** Commission digestion tasks and consume the Digest Agent's findings when making strategic decisions.
- **MicroAgentStacks:** Used to parallelise large digests—each stack handles a set of sources and feeds results back to the Digest Agent.
- **ModelSelectorAgents:** Select embedding models and summarisation LLMs appropriate for each source type. For example, code summarisation may use a codex model, while plain text summarisation uses a general LLM.
- **Data & Storage layer:** Stores artefacts and indices in MinIO, Postgres and the vector store. The Digest Agent ensures proper metadata tagging and retention policies.

By systematically consuming and analysing every relevant piece of information, the Digest Agent turns unstructured data into actionable knowledge for ark-os-noa's decision makers. # Backend — Services & Infrastructure of ark-os-noa

Purpose

The **backend** of ark-os-noa comprises all of the runtime services and infrastructure that turn high-level plans into concrete work. It includes the event bus, microservices that implement the **Expanded Digest Pipeline**, sidecars that enable the **Capsule** pattern, and internal data stores. Together, these components provide a robust, scalable and secure environment for executing tasks, orchestrated by NOA and the Board Agents.

Services & Microservices

Core Pipeline Services

The digest-everything pipeline is decomposed into a series of microservices, each responsible for a discrete stage. Running them as independent services ensures that each can scale, fail and be updated independently, which is aligned with microservice best practices 【43537238352704†L1068-L1088】.

1. **Intake Service:** Receives digest requests; validates inputs (repo URLs, API endpoints, model lists); creates provenance records and initializes workspace directories.

2. **Classifier Service:** Detects programming languages, build systems, service types (CLI, API, library) and licences. Produces a `profile.json` summarising the source.
3. **Graph Extract Service:** Parses code and schemas to build call graphs, data flow graphs and config surfaces. Supports multi-language parsing (Python, JS/TS, Go, Rust, Java). Outputs `kg.json` and `system_card.md`.
4. **Embeddings Service:** Generates embeddings for code and documentation using models selected by ModelSelectorAgents. Upserts vectors to pgvector or Qdrant.
5. **Env Synthesis Service:** Emits Dockerfiles, docker-compose YAML, Kubernetes manifests, `.env.example`, `Makefile` targets and config schemas. Ensures reproducible builds using outer BuildKit (no DinD).
6. **Safety Service:** Runs SBOM generation (Syft), vulnerability scans (Grype/Trivy), secret scans (Gitleaks) and static analysis (Semgrep). Applies policy gates; stops the pipeline on critical issues.
7. **Runner Service:** Builds and runs the source in a controlled container environment; executes existing tests or generates smoke tests; produces `demo.md`.
8. **Integrator Service:** Generates adapters (SDKs for Python, Node, Go), telemetry hooks and policy stubs; prepares packaging instructions.
9. **Registrar Service:** Writes outputs and metadata to storage (registry, MinIO, Postgres); registers embeddings; updates indexes for search.

Auxiliary Services

- **CRM Strangler Proxy:** Provides a transparent layer between internal clients and an external CRM. It records requests/responses, supports *shadow* and *write-through* modes, and allows incremental internal re-implementation of CRM features.
- **Model Serving:** Hosts local models using frameworks like llama.cpp, Ollama or vLLM. Exposes endpoints for inference and embedding generation. Each model server is packaged in its own container with health checks.
- **Gateway API:** A FastAPI service exposing endpoints: `/digest`, `/capsule/spawn`, `/crm/toggle`, `/models/ingest`, `/models/benchmark`, `/admin/*`. Acts as the single entry point for external clients and the front-end.

Event Bus & Orchestration

- **Redis Streams:** Provides the primary event bus for inter-service communication. Services consume and produce events in a decoupled fashion. The bus also supports message persistence and backpressure.
- **NATS (optional):** A lightweight publish/subscribe system for high fan-out or cross-cluster communication. Enabled via a feature flag.
- **Workflow Engine:** A simple DAG engine built on Redis to coordinate pipeline tasks with retries and backoff. Temporal or Argo Workflows can be integrated later for more sophisticated orchestrations.

Capsule Sidecars

All containers run inside a “Capsule” to simulate container-in-container and Kubernetes-in-Kubernetes workflows without the security and performance drawbacks 【716409907369096†L1037-L1067】. Capsule sidecars include:

1. **Build-Proxy:** A lightweight service that proxies inner `docker build` and `nerdctl` commands to the outer BuildKit daemon. It exposes a local socket inside the Capsule but forwards build requests externally, avoiding duplicate layer storage.
2. **Service-Mirror:** Watches inner service definitions and publishes corresponding services in the outer service mesh with mTLS and SLO configurations. This allows inner services to be reachable and observable from the outer plane.
3. **Policy Agent (OPA):** Enforces egress rules, resource quotas, and other policies at the Capsule boundary. It integrates with eBPF to block unauthorised traffic.
4. **Telemetry Agent:** Collects traces, metrics and logs from the inner services and sidecars. It forwards data to the central observability stack with proper trace-ID propagation.
5. **vcluster (optional):** Provides a lightweight Kubernetes API server inside the Capsule for tools that require kubectl. It maps pods to the parent cluster’s nodes without duplicating container runtimes.

Data Stores

- **OCI Registry:** Stores container images, compiled outputs and Capsule definitions. The registry uses content-addressed storage and enforces immutable tags.
- **MinIO:** Stores large artefacts, zipped deliverables, SBOMs and data sets. Supports versioning and server-side encryption.
- **Postgres (+ Supabase):** Maintains metadata (profiles, system cards, run logs), traces, job statuses and vector search indices. Supabase provides developer APIs and pgvector integration.
- **Vector Store:** For embeddings. The backend can be `pgvector` in Postgres or an external Qdrant instance. A feature flag chooses which driver to enable.

Security & Compliance

The backend enforces numerous policies:

- **No DinD:** Build operations are forwarded to outer BuildKit/containerd; containers run with user namespaces and seccomp, preventing container-root escalation 【435372383527 04†L1068-L1088】.
- **Licence & vulnerability gates:** The Safety service halts builds on critical issues; the Board Agents define accepted licence lists and vulnerability thresholds.
- **Secrets management:** Secrets are never stored in environment variables. They are mounted as files via Vault or similar systems, and sidecars are responsible for retrieving them.
- **Audit trails:** Every API call, pipeline event and model selection decision logs context (who, what, when, rationale). These logs live in Postgres and are tied to run IDs.

Development & Testing

- **Makefile:** Provides convenience targets (`make up`, `make down`, `make logs`, `make demo`, `make scan`, `make lock-images`) for developers. It ensures consistent environment setup and teardown.
- **Docker-Compose:** Defines services and dependencies; profiles enable optional components like NATS, Supabase and vcluster. Compose is used for local development. For production, manifests under `k8s/` can be applied to a Kubernetes cluster.
- **Automated tests:** Unit and integration tests run within the Runner Service; security scanners run in the Safety Service. CI pipelines (to be implemented post-launch) build images, run tests, generate SBOMs and publish artefacts.

By modularising the backend into clear services and infrastructure layers, ark-os-noa achieves the flexibility of microservices with the discipline of reproducible builds and strong security controls. # Data & Storage — Securing the ark-os-noa Data Plane

Principle

The data layer of ark-os-noa is built around a core principle: **keep all storage within the trust boundary**. All artefacts—images, datasets, logs, metadata, SBOMs—are retained internally, signed and versioned. Only signed, approved deliverables are exported. This ensures confidentiality, integrity and provenance across the platform.

Components

1. **Private OCI Registry:** Hosts container images, Capsule definitions, build outputs and adapters. Using a private registry prevents untrusted images from entering the environment and allows BuildKit to push/pull from a controlled backend.
2. **MinIO (S3-compatible object store):** Serves as the main artefact store. It holds large files (zip archives, compiled PDFs), dataset fragments, SBOM documents, vulnerability reports and even model shards. MinIO offers versioning, server-side encryption (SSE) and lifecycle management.
3. **Postgres:** Stores structured metadata: run logs, profiles (`profile.json`), system cards (`system_card.md`), call graphs (`kg.json`), job statuses, policy decisions and audit trails. Postgres also stores vector embeddings via the `pgvector` extension.
4. **Supabase:** A self-hosted instance of Supabase augments Postgres with developer APIs, authentication and real-time features. It provides a convenient interface for front-end applications and external tools until the platform fully internalises these capabilities.
5. **Vector Store:** Embeddings generated by the Embeddings Service are stored in either `pgvector` (Postgres) or a dedicated Qdrant cluster. pgvector is enabled by default for simplicity; Qdrant can be turned on via a feature flag to support larger vector workloads.

Policies & Best Practices

- **Immutability:** Artefacts are stored content-addressed using SHA-256 digests. Tags or names are pointers to immutable content; rewriting tags triggers new versions. This

prevents tampering and ensures reproducible builds.

- **Lineage & Provenance:** Each deliverable (zip, PDF, image) links back to its inputs, tools, models and run ID. Build provenance is stored as JSON attestation, capturing the environment, command, dependency versions and commit hashes.
- **Retention:** Short-lived runs (e.g. experimental digests) are kept for a limited period; long-term artefacts (e.g. official releases) are retained indefinitely. Policies can be configured per project or domain.
- **Access Control:** The data plane uses least privilege. Microservices receive temporary scoped tokens to access the object store or registry; access is audited. Secrets (e.g. tokens, keys) are stored in a secrets manager (Vault) and mounted as files, never as environment variables.

Integration with Other Components

- **Backend services:** Interact with the registry and MinIO via signed URLs or direct API calls. BuildKit pushes images to the registry; the Registrar Service writes artefacts to MinIO and records metadata in Postgres.
- **Digest Agent:** Reads and writes to MinIO and Postgres; uploads embeddings to the vector store. It uses the registry to store intermediate build images.
- **Model Selector and Model Servers:** Use Postgres (via pgvector or Qdrant) to store model metadata and evaluation results. Models themselves may be stored as OCI artefacts or in MinIO shards.
- **Front-end:** Accesses Supabase for real-time updates and uses signed URLs to fetch artefacts from MinIO.

One-Liners & Conventions

```
# Create local directories mirroring services (for dev/testing)
mkdir -p storage/{oci,minio,postgres,supabase,artifacts} && tree -L 2 storage
|| ls -R storage

# Content-address an artefact
digest=$(sha256sum output.zip | awk '{print $1}')
cp output.zip storage/artifacts/${digest}.zip
```

Why Internal Data Planes Matter

Keeping storage internal reduces the attack surface and simplifies compliance. Data never leaves the environment without explicit signing and approval. When combined with provenance tracking, this approach ensures that every piece of data can be traced back to its origin and verified—critical for regulated environments and supply-chain integrity. # Combined Framework & Architecture of ark-os-noa

High-Level Overview

ark-os-noa is an **agentic AI platform** designed to realise ElementArk/DeFlex's business model. It combines hierarchical organisational patterns (NOA → Board Agents → MicroAgentStacks →

microservices) with modern infrastructure techniques (Capsule/Full-Illusion pattern, private data plane, event bus) and an adaptable AI layer (ModelSelectorAgents and Digest Agent). The result is a "**hive mind**" of specialised agents capable of digesting, reasoning about and producing artefacts across software, data and SaaS systems.

Layers & Hierarchy

1. Strategy & Orchestration Layer

- **NOA:** The ExecutiveCommanderChiefAgent at the top. Transforms business goals into actionable plans, assigns Board Agents, sets policies, and monitors execution.
- **Board Agents:** Domain-specific executives (Strategy/CTO, COO, CFO/FinOps, Legal/Compliance, Security, Growth/Partnerships, Digest). Each can commission work via MicroAgentStacks and request ModelSelector assistance.

2. Execution Layer

- **MicroAgentStacks:** On-demand work pods orchestrated by a CommandChiefAgent. Each stack contains Operators, Adapters and Guards and runs through a defined lifecycle (Bootstrap → Execute → Validate → Package → Archive). Stacks interact with external sources (repos, CRMs, APIs) and internal services via Adapters.
- **Expanded Digest Pipeline:** A set of microservices (Intake, Classifier, Graph Extract, Embeddings, Env Synthesis, Safety, Runner, Integrator, Registrar) that perform the actual work. Each is loosely coupled via an event bus and runs inside the Capsule environment. CRM Strangler and Model Serving are additional services.

3. Infrastructure Layer

- **Capsule Architecture (Full Illusion):** Encapsulates stacks and services in a sandbox that forwards build operations and network traffic to the outer runtime. Capsule sidecars (Build-Proxy, Service-Mirror, Policy Agent, Telemetry Agent, optionally vcluster) provide the illusion of Docker-in-Docker and Kubernetes-in-Kubernetes without their drawbacks【716409907369096†L1037-L1067】.
- **Event Bus & Orchestration:** Redis Streams (primary) and optional NATS enable asynchronous communication. A workflow engine coordinates the pipeline steps, handling retries and backoff.
- **Data Plane:** Private OCI registry, MinIO, Postgres (+ pgvector/Supabase) and optionally Qdrant. This plane stores everything from container images to embeddings and ensures data stays within the trust boundary.
- **Observability & Security:** OTel tracing, Prometheus metrics, policy agents, SBOM/vulnerability scanners and secrets management. The **no DinD** policy and user namespaces reduce privilege escalation risk【43537238352704†L1068-L1088】.

How the Pieces Fit Together

1. **Goal Intake:** A high-level goal arrives. NOA normalises it into a WorkPlan and determines which Board Agents are responsible.
2. **Board Planning:** Board Agents refine the goal, assign budgets, define SLAs and set policies. They request MicroAgentStacks and ModelSelectorAgents as needed.
3. **Stack Deployment:** For each task, a MicroAgentStack is spawned. The stack uses Adapters to fetch sources (repos, CRMs), Operators to parse/analyse, and Guards to enforce policies. Microservices implement the digest pipeline, orchestrated via the event bus.
4. **Model Selection & Execution:** When a service or operator needs AI inference (embeddings, summarisation, code explanation), it calls a ModelSelectorAgent. The selected model is executed via local model servers or remote APIs.
5. **Data Persistence:** Outputs from each step (SBOMs, graphs, embeddings, demos) are persisted via the Data Plane. The Registrar Service updates indexes and metadata.
6. **Completion & Reporting:** Once tasks finish, the stack packages results into a zip and compiled PDF, publishes them to MinIO and the registry, and updates Postgres. NOA receives a report and archives the run.

Why This Architecture?

1. **Modularity & Scalability:** By decomposing functionality into microservices and agents, ark-os-noa can scale horizontally and update components independently—avoiding the pitfalls of monolithic systems【43537238352704†L1068-L1088】.
2. **Security & Compliance:** The Capsule pattern, no DinD policy, private data plane and sidecar enforcement minimise the attack surface. SBOMs, licences and vulnerability scans ensure supply-chain integrity.
3. **Intelligence & Adaptability:** ModelSelectorAgents enable adaptive AI usage; the Digest Agent builds knowledge graphs and embeddings; the board can ingest CRMs and SaaS systems without downtime using the strangler proxy.
4. **Auditability & Provenance:** Every decision, model selection and action is logged in Postgres and associated with a run ID. Artefacts are content-addressed and signed. This supports post-mortems, compliance and future learning.

Extensibility

- **New Board roles:** Additional executives (e.g. Marketing, HR) can be added by extending the roster and defining their domains and policies.
- **Additional microservices:** New processing stages (e.g. code transformers, simulation engines) can be plugged into the pipeline without redesigning the whole system.
- **Hybrid deployment:** While Compose is used locally, Kubernetes manifests (`k8s/`) can be applied to a cluster; the same Capsule pattern applies.
- **Model & Connector expansions:** New AI models are registered via the ModelSelector; new connectors are implemented by Adapters to integrate more SaaS or data sources.

The **Combined Framework & Architecture** unifies strategic planning, microservice execution,

security and AI into a cohesive system. It is intentionally modular to allow continuous growth and improvement. # API, Connectors & Front-End of ark-os-noa

Gateway API

The **Gateway API** is the central entry point for interacting with ark-os-noa's backend services. Implemented using FastAPI, it exposes endpoints for ingesting sources, spawning capsules, toggling CRM behaviours, ingesting models and administering the system.

Key Endpoints

Endpoint	Method	Description
`/digest`	POST	Submit a digest request. The request includes sources (e.g. repo URL, API base URL), intent (integrate, analyse), and optional metadata. It triggers the Intake Service and returns a job ID.
`/capsule/spawn`	POST	Spawn a new Capsule environment. Returns Capsule identifiers and access tokens. Used when custom stacks need to be run manually or via the front-end.
`/crm/toggle`	POST	Toggle the CRM Strangler Proxy mode for a specific endpoint (e.g. enable write-through for `/contacts`). Allows incremental migration from external CRM to internal implementation.
`/models/ingest`	POST	Add a model to the local registry. Accepts a model identifier (e.g. Hugging Face repo) and optional metadata. The Model Serving service pulls the model and makes it available through the ModelSelector.
`/models/benchmark`	POST	Run evaluations on local or remote models. Returns latency, cost and accuracy metrics that feed into the ModelSelector's decision graph.
`/admin/*`	GET/POST	Administrative endpoints for tasks such as inspecting job statuses, viewing SBOMs, retrieving logs, enabling/disabling features (NATS, Supabase, vcluster) and rotating secrets. Protected via authentication and authorisation.

All endpoints accept and return JSON; error responses include descriptive messages and relevant

codes. The Gateway uses request identifiers and attaches trace IDs to facilitate debugging and correlation across services.

Connectors & Integrations

ark-os-noa interacts with the outside world via **Adapters** and **Connectors**. These modules encapsulate authentication, rate limiting, and protocol details, allowing the rest of the system to remain agnostic to third-party specifics.

Built-in Connectors

- **GitHub Connector:** Uses the GitHub API to search, clone and pull repositories. It supports scoping by organisation or repository and can read commit logs and PR metadata.
- **CRM Connector:** Provides read/write access to CRM systems (e.g. Salesforce, HubSpot). Initially operates in shadow mode (read-only) via the CRM Strangler Proxy; write-through can be toggled per endpoint. Handles pagination, rate limits and authentication.
- **Model Hub Connector:** Interfaces with external model repositories (e.g. Hugging Face). Supports pulling models, downloading tokenizers and retrieving licences. Works in conjunction with the Model Serving service.
- **Other API Connectors:** Additional connectors (e.g. for Slack, Notion, Jira) can be added by implementing the Adapter interface. Each connector is packaged as its own microservice or plugin to preserve modularity.

Internal Connectors

- **Registry & Object Store:** Adapters communicate with the private OCI registry and MinIO using signed URLs. They ensure that images and artefacts are pushed/pulled securely and that content addressing is respected.
- **Database & Vector Store:** Adapters abstract database interactions. They provide typed functions to query or insert metadata, run logs and embeddings without exposing SQL directly to the application logic.

Front-End (Admin Console)

The **Admin Console** is a web interface built with Next.js. Its primary function is to give administrators and power users visibility and control over the system. Major features include:

- **Jobs Dashboard:** Displays active and past digest jobs, their statuses, progress bars and any errors. Users can drill down into individual jobs to view their `profile.json`, `system_card.md`, SBOMs and vulnerability reports.
- **Capacities & Capsules:** Shows currently running Capsules, their resource usage and health status. Offers controls to spawn or destroy Capsules.
- **Artefacts Explorer:** Lists generated artefacts (zip files, PDFs, embeddings, SBOMs). Allows downloading via signed URLs and cross-referencing to their origins.

- **SBOM & Security:** Provides a dedicated section to review SBOMs, vulnerabilities, licences and risk scores. Policies can be configured here (e.g. accepted licence list, vulnerability severity thresholds).
- **Model Registry & Selector:** Displays available models, their metadata, benchmarks and usage statistics. Administrators can add models to the ingestion queue or deprecate existing ones. The ModelSelector's decisions and rationales are visible for transparency.
- **CRM Controls:** Allows toggling of CRM endpoint modes (shadow/write-through), viewing recent calls, and measuring divergence between external CRM data and internal state.
- **Settings & Feature Flags:** Provides toggles for enabling/disabling optional services (NATS, Supabase, vcluster) and adjusting environment variables. Also offers secret rotation and certificate management.

Interaction Patterns

- **External Clients:** Use the Gateway API to submit work. They receive job IDs and can query progress or results. Authentication tokens limit access based on roles.
- **Internal Agents:** Call endpoints via Adapters. For example, a CommandChiefAgent may call `/digest` to start digestion for a new source or `/models/ingest` to add an in-house model. Internal calls attach run IDs and context for traceability.
- **Front-End Users:** Access the Admin Console to monitor and control the system. When they trigger actions (e.g. toggling a CRM endpoint), the console issues calls to the Gateway API on their behalf.

By exposing a clear API and a rich front-end, ark-os-noa ensures that humans and agents can seamlessly interact with the system, inspect its state and adapt its behaviour without compromising security or traceability. # Intelligence & Learning in ark-os-noa

Vision

ark-os-noa aspires to be more than an automation platform—it aims to embody **agentic intelligence**. Intelligence here means the ability to understand complex systems (codebases, data sets, SaaS integrations), reason about them, learn from past executions, anticipate future scenarios, and adapt models and workflows accordingly. Learning is achieved through a combination of semantic understanding (knowledge graphs and embeddings), model evaluation, feedback loops and simulation of alternative futures ("branchwise foresight").

Semantic Understanding

At the heart of ark-os-noa's intelligence lies a **semantic representation** of the world:

- **Knowledge Graphs:** Built by the Graph Extract and Digest services, these graphs link code symbols, data entities, API endpoints, configuration keys and other artefacts. They capture relationships (calls, imports, reads/writes, dependency edges) and annotate nodes with metadata (e.g. licence, language, risk). Knowledge graphs enable graph-based queries and reasoning—answering questions like "Which services write to table X?" or

"What code paths handle payment processing?"

- **Embeddings & Vector DB:** The Embeddings Service converts source code, documentation and natural-language descriptions into high-dimensional vectors. Stored in pgvector or Qdrant, these vectors power similarity search and clustering, enabling retrieval of semantically related items even if keywords differ.

Model Evaluation & Evolution

The **ModelSelectorAgent** plays a central role in learning. By recording the performance (latency, cost, accuracy) and outcomes of each model used for a task, the system builds a knowledge base of model behaviours. Over time, the selector's heuristics can be tuned or even replaced by learned policies that maximise utility subject to constraints. Benchmark results and feedback loops allow the system to retire underperforming models and onboard new ones seamlessly.

Feedback Loops & Trace Learning

Every execution produces a **Trace**—a record of inputs, actions, decisions, outputs and outcomes. These traces are stored in Postgres along with logs and metrics. Post-run analyses mine these traces to identify patterns:

- **Success patterns:** Which workflows succeeded quickly with minimal retries? Which models performed best on certain task types?
- **Failure modes:** Which tasks frequently hit policy violations or vulnerabilities? Which connectors are unreliable?
- **Cost hot-spots:** Where is budget being spent? Are there cheaper alternatives?

Insights from these analyses can feed back into NOA's planning and ModelSelector policies, closing the loop between execution and learning.

Mind Maps & Branchwise Foresight

The system leverages the knowledge graph and embeddings to construct **mind maps**—visual or conceptual maps of relationships between components, tasks and dependencies. These maps assist in reasoning about the impact of changes, identifying missing connections and planning new integrations.

Branchwise foresight refers to simulating multiple potential futures or scenarios before committing resources. For example, before migrating a CRM function internally, NOA can instruct a MicroAgentStack to:

1. **Simulate Strategy A:** Keep the external CRM; use the strangler proxy in shadow mode; measure divergence.
2. **Simulate Strategy B:** Implement a minimal internal replacement for a specific endpoint; run synthetic load; compare latency and correctness.
3. **Simulate Strategy C:** Replace the CRM entirely with internal modules and measure performance, cost and user impact.

By comparing the outcomes of these branches, NOA and the Board Agents can choose a course of action informed by data rather than intuition. This approach aligns with the idea of **compound AI systems**, where tasks are decomposed into specialised modules and their outputs orchestrated 【438618440126565†L248-L292】.

Continuous Learning & Improvement

Learning in ark-os-noa is continuous:

- **Auto-patch loops:** When tests fail, Graph Extract proposes diffs, Runner applies them, and Safety verifies the fixes. Successful patches can be proposed back to source repositories as pull requests.
- **Change intelligence:** Scheduled self-digests detect changes in upstream sources; the system predicts breaking changes and generates migration guides.
- **Policy refinement:** The Board and NOA adjust policies (licence lists, vulnerability thresholds, model selection heuristics) based on operational data and emerging requirements.

By combining semantic representations, model analytics, feedback loops and foresight simulations, ark-os-noa evolves beyond a static workflow runner into an adaptive system capable of strategic reasoning and self-improvement.