AgentDock PRD

1. Overview & Vision

1.1 Product Summary

AgentDock is an **Agent Deployment and Operations Platform** that enables AI engineers to turn locally developed LLM/agent workflows—built with frameworks like LangGraph or LangChain—into **secure, observable, and governed microservices** with a single declarative configuration file called a **Dockfile**.

In short:

> "AgentDock turns agent code into a production-ready API with one command."

Each Dockfile defines:

- The agent's **entrypoint** (e.g., a LangGraph graph builder)
- Its **model + runtime configuration**
- **Input/output schema**
- **Auth, policy, and observability settings**

AgentDock automatically:

1. Parses the Dockfile,
2. Generates a FastAPI runtime with validated endpoints,
3. Wraps the agent in containerized infrastructure with security and telemetry,
4. Deploys it as a stand-alone service available via REST or streaming APIs.

1.2 Problem Statement

Current Pain Points

Building intelligent agents has become easy; **productionizing them is not**.
Developers today must manually:

- Wrap agents with API layers
- Manage authentication and authorization
- Inject secrets safely

- Collect logs and token cost metrics
- Deploy via Docker/K8s and maintain observability dashboards

This process requires infrastructure and DevOps expertise most AI teams lack, leading to:

- Inconsistent security and governance
- Limited visibility into cost and latency
- Difficult agent versioning and rollback
- Slow iteration from prototype → production

1.3 Product Vision

AgentDock's long-term mission is to become the **control plane for autonomous agent ecosystems**—a unifying layer that manages **deployment, security, observability, and inter-agent communication** across frameworks and environments.

Vision Timeline

Phase	Tagline	Primary Goal
v1	*"Docker for Agents"*	Simplify single-agent deployment & monitoring
v2	*"AgentOps Platform"*	Provide registry, versioning, RBAC, cost dashboards
v3	*"Agent Mesh Cloud"*	Orchestrate distributed multi-agent systems with governance & billing

---

1.4 Scope of v1

In Scope

- **Single-agent deployments** built on LangGraph or LangChain
- **Dockfile v1 Spec:** agent metadata, schemas, policies, auth, observability
- **Automatic API exposure:** REST + SSE/WebSocket
- **AuthN/AuthZ:** JWT, API keys, basic roles
- **Telemetry:** LangFuse integration for latency, token usage, cost
- **Containerized runtime:** Docker + Kubernetes deployments
- **Web dashboard:** status, logs, metrics, keys, versions

Out of Scope (for v1 but planned)

- Multi-agent workflows or meshes
- Fine-grained RBAC and organization-wide policies
- Billing and token-based quotas
- Marketplace / registry of agents
- Shared multi-tenant runtime

1.5 Success Metrics (KPIs for v1)

KPI	Target	Measurement Method
Deployment Time	≤ 60 seconds from Dockfile → live API	CLI logs / deployment metrics
Setup Reduction	≥ 80 % less manual infra setup than traditional FastAPI deploy	Developer survey / time tracking
Latency Overhead	< 10 % vs raw LangGraph runtime	Load testing
Telemetry Coverage	100 % runs captured in LangFuse with tokens/cost	LangFuse dashboard sync
Error Containment	< 2 % failed runs per 1000 requests	AgentDock logs
Developer Satisfaction	> 8/10 post-pilot	Internal feedback survey

---

v1 Goal Summary

Deliver a **robust, developer-first MVP** that:

- Eliminates manual DevOps steps,
- Establishes a reliable deployment contract (Dockfile + runtime),
- Demonstrates enterprise-grade security and observability patterns,
- Provides a scalable foundation for the AgentOps platform (v2) and the future multi-agent mesh (v3).

---

2. Target Users & Personas

2.1 Primary Persona – AI / ML Engineer (Agent Builder)

Profile

- Technical professional developing LLM-powered agents or workflows using LangGraph, LangChain, or DSPy.
- Strong in Python and model experimentation, weak in DevOps or production engineering.
- Works in small teams or innovation pods within enterprises.

Pain Points

- Has functional agent code but struggles to deploy it securely.
- Manual setup for API, auth, and telemetry is repetitive.
- No standardized way to version or monitor agents.
- Needs a platform that converts prototypes into usable endpoints.

Goals with AgentDock

- Deploy agent directly from Dockfile to a live REST API.
- Auto-generate OpenAPI docs, manage auth tokens, and track usage metrics.
- Avoid manual infra and repetitive configuration.
- Validate that the deployed agent behaves consistently.

2.2 Secondary Persona – Platform / DevOps Engineer

Profile

- Manages AI infrastructure and deployment pipelines.
- Responsible for uptime, scaling, and compliance of agent services.
- Works closely with data and AI teams in mid–large enterprises.

Pain Points

- Developers build agents that are hard to monitor or secure.
- Re-deployment, policy updates, and scaling are manual.
- Needs a standardized, auditable process for deploying agents across teams.

Goals with AgentDock

- Standardize deployment workflows using Dockfile contracts.
- Enforce security, auth, and monitoring policies automatically.
- Visualize agent health, latency, and cost metrics.
- Integrate with enterprise CI/CD and observability stacks.

2.3 Future Persona – Enterprise Admin / Org AI Lead

Profile

- Oversees organization-wide AI deployment governance.
- Interested in ROI, data security, cost optimization, and compliance.
- Uses dashboards to audit usage across business units.

Future Needs (v2/v3 relevance)

- Manage org-wide policies, API key governance, and team-level access.
- View multi-agent telemetry, cost reports, and compliance summaries.
- Approve or restrict certain tools/models at policy level.
- Link AgentDock data into enterprise billing and cost-tracking systems.

2.4 User Journey Before and After AgentDock

Stage	Before AgentDock	After AgentDock
Build	Write LangGraph agent code manually, local testing only	Same process; just expose `build_graph()` function
Configure	Write FastAPI wrappers, env files, manual secrets	Write Dockfile once: define agent, model, auth, policy, telemetry
Deploy	Docker build, push, K8s manifest, logs setup manually	`agentdock deploy` → automatic containerization + FastAPI runtime
Access	Curl local API or manual Postman setup	Auto-generated REST/SSE endpoints with JWT/API keys
Monitor	Manual print logs, no cost tracking	LangFuse dashboards for latency, cost, token usage
Iterate	Manual redeploys, config drift	Versioned Dockfile → instant rollback and re-deploy
Scale	Handled by Ops team	Policy-based auto-scaling (v2)

Key Insight

AgentDock's user experience should feel like:

> "I only need to focus on building my agent's logic — everything else (deploy, auth, metrics, docs) is handled by the platform."

---

3. Product Goals & Non-Goals

3.1 Product Goals (for v1)

G-1 Unified Deployment Contract

Provide a single declarative **Dockfile (YAML)** that completely defines how an agent is deployed — entrypoint, model, schema, policies, auth, telemetry, and exposure settings.

- ✅ Outcome: Developers need zero DevOps scripts to get a secure, observable API.

G-2 Automated Runtime Generation

Generate a **FastAPI-based runtime** automatically from the Dockfile.

- Parse Dockfile → create endpoints (`/invoke` , `/stream` , `/health`)
- Build OpenAPI 3.0 docs dynamically
- Attach authentication and validation middlewares
- Wrap agent logic (LangGraph adapter v1)

G-3 Security & Governance Defaults

Every deployed agent must have production-grade security by default.

- JWT and API-key support
- Tool allow/deny policies
- PII redaction patterns
- Role-based access control (minimal roles for v1)

G-4 Observability & Telemetry

Integrate with **LangFuse** to track:

- Latency (p50/p90/p99)
- Token usage and cost
- Error rates and retries
- Tool-level invocation time

Provide a minimal dashboard showing these metrics per agent.

G-5 Versioning & Rollback

Each deployment = a **versioned artifact**.

- Store Dockfile and runtime metadata
- Support `rollback <agent> <v>`
- Track who deployed which version and when

G-6 Developer Experience (UX & CLI)

Offer an intuitive **CLI / API surface** for:

```
agentdock init
agentdock validate
agentdock deploy
agentdock rollback
agentdock status
```

Each command must emit structured logs suitable for CI/CD integration.

G-7 Scalable Architecture Foundation

Lay groundwork for future multi-agent and multi-tenant expansion:

- Adapter interface for LangGraph → DSPy/Autogen later
- Container-per-agent runtime (isolated pods)
- Metadata DB (Postgres) with org and version tables

G-8 Minimal Web Dashboard

Ship a lightweight React dashboard for:

- Listing deployed agents
- Viewing status, latency, cost, token usage
- Viewing API docs / schema
- Rolling back versions
- Managing API keys

3.2 Non-Goals (for v1)

Area	Deferred Feature	Rationale / Planned Phase
Multi-Agent Workflows	Cross-agent routing / mesh	Planned v2–v3 (after stable single-agent runtime)
Advanced RBAC	Org/team hierarchies, fine-grained policies	Simplify v1 to admin/dev/viewer roles
Billing & Quotas	Token or usage-based billing	Introduce post MVP once telemetry is solid
Marketplace / Registry	Agent sharing and discovery	v2 feature (AgentOps Platform)
Auto-Scaling and Job Queues	Dynamic worker provisioning	Requires metrics stability first
Multi-Framework Adapters	DSPy, Autogen, Semantic Kernel	LangGraph only for v1
Enterprise Integrations	SSO, SOC2, Cloud Billing	v2+ enterprise targets
UI Design System	Full UX polish and theme library	MVP UI only, focus on functionality
Multi-Tenant Runtime	Shared agent workers	Keep container-per-agent for v1 clarity
Marketplace Plugins or SDKs	3rd-party extensions	Add in v2 once core API is frozen

---

3.3 Success Definition for v1

AgentDock v1 is successful when:

- An AI engineer can deploy a LangGraph agent from Dockfile → live API in ≤ 60 seconds.
- Telemetry data and auth work out-of-the-box without manual config.
- Rollback and OpenAPI generation operate flawlessly.
- Platform runs reliably for 1000+ requests/day per agent with < 10 % overhead.

---

4. System Overview

This section gives your team a precise “big picture” of AgentDock v1—what runs where, how pieces talk, and the lifecycle from a Dockfile to a production API.

4.1 High-Level Architecture (at a glance)

Control Plane (one): CLI, Config/Schema engine, Builder/Packager, Metadata DB, Dashboard, Key/Policy services
Data Plane (many): One **container/pod per agent** running auto-generated FastAPI runtime + Adapter (LangGraph v1)

4.2 Core Components (v1)

1) CLI

- Commands: `init`, `validate`, `deploy`, `status`, `rollback`
- Validates Dockfile, streams build/deploy logs, returns IDs/versions (for CI/CD)

2) Config Parser & Schema Engine

- Parses **Dockfile v1** → Pydantic models
- Accepts schema in YAML *or* Python class paths (`io_schema.input_class`)
- Enforces required fields, policy format, and exposure constraints

3) Builder/Packager

- Generates **FastAPI runtime** from Dockfile
- Emits OpenAPI spec, mounts middlewares (auth, redaction)
- Builds Docker image (per agent version), pushes to registry (optional)
- Produces deployment manifest (Docker/K8s)

4) Agent Runtime Adapter (LangGraphAdapter v1)

- Loads `entrypoint` (e.g., `app.graph:build_graph`) → returns a graph
- Invokes/streams graph with validated input
- Normalizes events for telemetry & logs

5) Policy & Auth Layer

- **AuthN**: JWT + API keys (v1)
- **AuthZ**: minimal roles (admin/dev/viewer); tool allow/deny
- **Safety**: regex redaction; optional output checks (basic)

6) Telemetry Module

- Integrations with **LangFuse** for runs, tokens, latency, cost
- Emits structured events per tool/node
- Local aggregation (fallback) for air-gapped deployments

7) API Gateway (FastAPI Runtime)

- Auto-generated endpoints:
  - `POST /invoke` (sync)
  - `GET /stream` (SSE/WebSocket)
  - `GET /health`
  - `GET /schema` (I/O)
- Swagger UI for live testing

8) Metadata Store (Postgres)

- Tables for: agents, versions, deployments, API keys, roles, policies
- Links run IDs to LangFuse trace IDs
- Stores Dockfile snapshots (for rollback)

9) Dashboard (React/Next.js)

- **Agents list**: status, current version, last deployer
- **Metrics**: p50/p90 latency, tokens, cost, error rate
- **Docs**: embedded Swagger / OpenAPI viewer
- **Ops**: rollback, key rotation, policy view

4.3 Lifecycle / Data Flow (Dockfile → Live API)

1. **Authoring**
   - Dev writes/updates LangGraph agent and Dockfile (`entrypoint` , `io_schema` , `policies` , `auth` , `observability` , `expose` , `arguments`).
2. **Validation**
   - `agentdock validate`
   - Parser checks YAML shape; Schema Engine builds Pydantic models or imports Python classes; warns on missing/unsafe policies.
3. **Build**
   - `agentdock deploy`
   - Builder generates FastAPI app (routes + middlewares), injects Adapter, bakes OpenAPI, packages Docker image.
4. **Register**
   - Control Plane writes **agent + version** metadata to Postgres; stores Dockfile snapshot; tags image.
5. **Run**
   - Data Plane starts container/pod. FastAPI boots, loads LangGraph via Adapter, publishes health.
6. **Expose**
   - Endpoints become available: `/invoke` , `/stream` , `/schema` , `/health` .
   - Dashboard shows **Running** with version & OpenAPI link.
7. **Observe**
   - Each request emits **LangFuse** traces (timings, token usage, cost, tool spans).
   - Dashboard charts update.
8. **Iterate / Rollback**
   - New Dockfile → new version → blue/green replace (v1 can be simple stop/start).

- Rollback picks snapshot + previous image → restart.

4.4 Primary Interfaces (clean seams)

CLI ↔ Control Plane API

- `POST /deploy` → payload: Dockfile, repo ref, env
- `GET /status?agent=name` → returns version/state/log tail
- `POST /rollback` → `{ agent, version }`

Control Plane ↔ Data Plane

- **Deploy manifest** provisioned (Docker/K8s)
- Health/readiness checks (HTTP 200, version header)
- Signed config bundle (policies, keys) mounted at startup

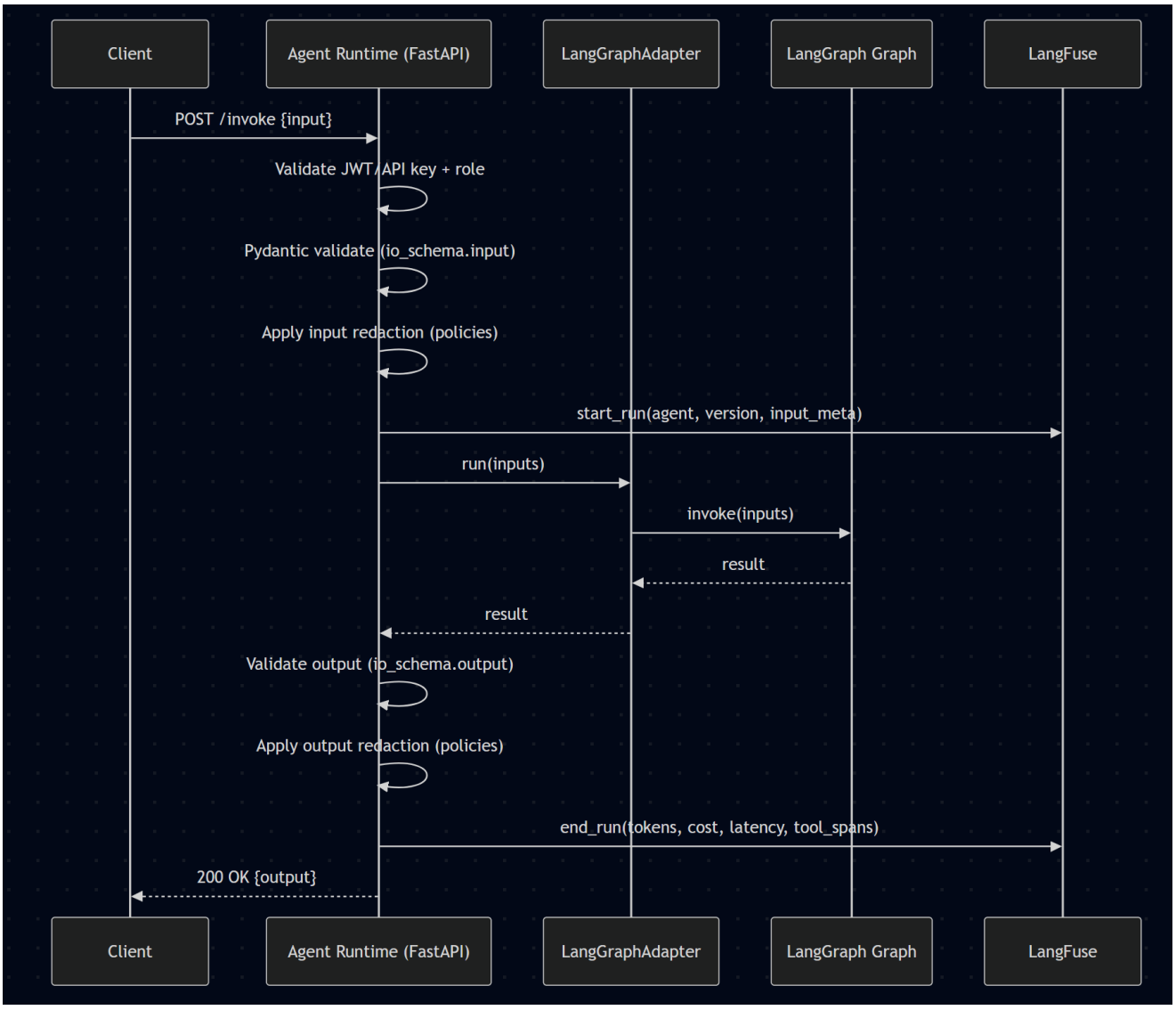
Runtime ↔ Adapter

```
class AgentAdapter:
    def load(self, entrypoint: str, config: dict): ...
    def run(self, agent, inputs: dict) -> dict: ...
    def stream(self, agent, inputs: dict) -> Iterable[dict]: ...
```

Runtime ↔ Policy/Auth Middlewares

- `before_invoke` : verify JWT/API key, role, tool gating, redact input
- `after_invoke` : redact output (if configured), emit metrics

4.5 Sequence Diagram (invoke path)



4.6 Deployment Topologies (v1)

- **Local Dev**: Docker Compose (Control Plane + one Agent Runtime)
- **Small Team / Demo**: Control Plane on Render/Vercel + agent pods on Render/Azure App Service
- **Enterprise**: Control Plane in VPC (K8s), agents as **one pod per agent** (Helm), Postgres managed (Azure PG), LangFuse self-hosted or SaaS

Why per-agent pods? Isolation, independent scaling, version parity, simpler rollback.

4.7 Minimal Data Model (Postgres)

agents

`id, name, created_at, owner, org_id`

agent_versions

`id, agent_id, version, dockfile_yaml, image_ref, created_at, created_by`

deployments
`id, agent_version_id, status, endpoint_url, started_at, stopped_at`

api_keys
`id, agent_id, role, hashed_key, created_at, revoked_at`

policies
`id, agent_id, version_id, policy_json, created_at`

(Traces/tokens/cost primarily in **LangFuse**; store run ID references here if needed.)

4.8 Non-Functional Targets (anchor for Section 9)

- **Cold start** ≤ 5s (FastAPI + adapter)
- **Latency overhead** ≤ 10% vs raw graph
- **p90 invoke** ≤ 2.5s (model/provider dependent)
- **Availability** 99.5% (single zone), 99.9% (HA option)
- **Security**: keys hashed, JWT signed, TLS everywhere

4.9 Design Principles (guardrails)

- **Schema-first:** All endpoints driven by I/O schema → OpenAPI → SDKs.
- **Secure-by-default:** Auth on, tools gated, redaction enabled where relevant.
- **Pluggable adapters:** LangGraph now; DSPy/Autogen later without rewrites.
- **Stateless runtime edge:** Persist only metadata; push traces to LangFuse.
- **Observability always-on:** Every run emits structured spans.

5.1 Agent Deployment Lifecycle

5.1.1 Dockfile Parsing

Objective

Transform the user-supplied Dockfile (YAML v1 spec) into an internal, strongly-typed configuration object used by every downstream module.

Functional Flow

1. User executes `agentdock deploy --file Dockfile.yaml`.
2. CLI uploads file → Control Plane API.
3. Config Parser Module:
   - Loads YAML.
   - Validates against JSON-Schema v1 spec.
   - Expands environment variables (`${VAR}` syntax).
   - Normalizes relative imports (entrypoints, schema paths).
   - Constructs a `DockSpec` Python object.

Acceptance Criteria

- ✅ Invalid YAML → explicit parse error (`code = 4001`).
- ✅ Missing mandatory keys (`agent.name`, `entrypoint`, `model`) → schema error.
- ✅ Nested includes (e.g., `!include defaults.yaml`) supported.
- ✅ Parser produces normalized dict ready for Pydantic validation.

5.1.2 Validation (Stage 2)

Objective

Ensure the Dockfile describes a deployable, safe, and internally consistent agent.

Validation Layers

Layer	Check	Failure Response
Schema Validation	Matches Dockfile v1 JSON Schema	`InvalidSchemaError`
Entrypoint Validation	Path `<module>:<func>` exists and callable	`EntrypointNotFoundError`
I/O Schema	Imports Pydantic class or inline schema is valid	`InvalidIOSchemaError`
Policy	Tool allow/deny lists valid YAML types	`PolicyValidationError`
Auth Block	At least one auth mode enabled	`AuthConfigError`
Observability Block	Valid LangFuse keys if telemetry enabled	`TelemetryConfigError`

Acceptance Criteria

- ✅ CLI reports `✓ Validated successfully` on pass.

- ✅ Each validation error shows path (`auth.jwt.secret_key missing`).
- ✅ CI/CD pipelines can run `agentdock validate --strict` for non-interactive checks.

5.1.3 Container Build & Runtime Spin-Up

Objective

Package the validated agent into a container with an auto-generated FastAPI runtime and deploy it to the execution environment.

Build Flow

1. Builder Module creates temp workspace:
   - Injects FastAPI boilerplate (`main.py` , `requirements.txt` , `run.sh`).
   - Generates OpenAPI schema via Pydantic I/O classes.
   - Mounts auth, policy, and observability middleware templates.
2. Builds Docker image tagged `agentdock/<agent>:<version>` .
3. Pushes image (optional if local mode off).
4. Deploys container/pod through:
   - **Local mode:** Docker Compose service.
   - **Enterprise mode:** Kubernetes Deployment via AgentDock Controller.

Acceptance Criteria

- ✅ Container build completes ≤ 90 s for typical LangGraph agent.
- ✅ Runtime exposes ports `8080/443` with health endpoint.
- ✅ Deployment metadata written to Postgres (`agent_versions` , `deployments`).
- ✅ Logs stream to CLI during build/deploy.

5.1.4 FastAPI Auto-Generation

Objective

Convert the DockSpec into a fully functional FastAPI service with validated endpoints and observability hooks.

Endpoints

Endpoint	Method	Description
`/invoke`	POST	Executes agent synchronously with validated input.
`/stream`	GET	Streams token/events (via SSE or WebSocket).
`/schema`	GET	Returns JSON Schema for input/output models.
`/health`	GET	Health and version info.

Generation Steps

1. Create Pydantic models from I/O schemas.
2. Register routes and attach middlewares (auth, policy, telemetry).
3. Generate OpenAPI 3.0 spec and serve via Swagger UI.
4. Initialize LangGraphAdapter for entrypoint.

Acceptance Criteria

- ✅ `GET /schema` matches Dockfile I/O schema.
- ✅ Auth enforced on `/invoke` and `/stream`.
- ✅ Swagger UI available at `/docs`.
- ✅ Model runs and returns validated output.

5.1.5 Versioning & Rollback

Objective

Track every deployment as a versioned artifact and enable instant rollback to any previous state.

Process

1. Each `agentdock deploy` → new `version` record in Postgres.
2. Stores: `dockfile_yaml`, `image_ref`, `deployed_by`, `timestamp`.
3. Rollback command: `agentdock rollback <agent> <vX.Y>`
   - Stops current pod.
   - Pulls previous image + Dockfile.
   - Re-registers metadata.
4. Dashboard shows version history and rollback button.

Acceptance Criteria

- ✅ Versions are immutable once deployed.

- ✅ Rollback completes < 30 s on single agent.
- ✅ Post-rollback health check = green before marking “active”.

Lifecycle Completion Definition

> The Agent Deployment Lifecycle is complete when a developer can run `agentdock deploy` → a secure, observable, versioned agent API is live and invokable, without writing any additional infra or auth code.

5.2 Dockfile Specification (v1 Schema)

5.2.1 Purpose

The **Dockfile** is a YAML-based declarative configuration that defines how an agent should be deployed, secured, and observed.
It replaces the need for manual Dockerfiles, environment variables, and API scaffolding.
AgentDock reads this file, validates it, and automatically generates:

- A FastAPI runtime
- Auth & policy layers
- OpenAPI specs
- Telemetry integration
- Container deployment metadata

5.2.2 Top-Level Structure

```yaml
version: "1.0"                # Dockfile schema version
agent:
  name: invoice_copilot
  description: "Extract and summarize invoice data"
  entrypoint: app.graph:build_graph
  framework: langgraph          # Framework adapter (langgraph, langchain, etc.)

model:
  provider: azure
  name: gpt-4o
  temperature: 0.2
```

```yaml
  max_tokens: 2000
  endpoint: https://api.openai.azure.com

io_schema:
  input_class: app.schemas.InvoiceInput     # or inline YAML (see below)
  output_class: app.schemas.InvoiceOutput

arguments:
  max_retries: 2
  timeout_sec: 20
  enable_streaming: true

policies:
  tools:
    allowed: [extract_invoice, summarize_invoice]
    deny_by_default: true
  safety:
    redact_patterns:
      - "\d{16}"     # credit-card redaction
    max_output_chars: 5000

auth:
  mode: jwt
  jwt:
    secret_key: "${JWT_SECRET}"
    expiry_minutes: 60
  api_keys:
    enabled: true

observability:
  langfuse:
    project: invoice-copilot
    public_key: "${LANGFUSE_PUBLIC}"
    secret_key: "${LANGFUSE_SECRET}"
  tracing: true
  log_level: info

expose:
  rest: true
  streaming: sse
  port: 8080

metadata:
  maintainer: paritosh.sharma@company.com
  version: 1.0.0
  tags: [finance, extraction]
```

5.2.3 Mandatory Fields

Field	Type	Description
`agent.name`	string	Unique agent identifier
`agent.entrypoint`	string `<module>:<function>`	LangGraph or LangChain entry function
`model.provider`	enum (`azure` , `openai` , `anthropic` , etc.)	Defines model backend
`io_schema`	object	Input/output definition (YAML or class paths)
`auth`	object	At least one auth method must be active
`expose`	object	Defines runtime exposure (REST/SSE)

5.2.4 I/O Schema Definition

Option A – Inline YAML

```yaml
io_schema:
  input:
    type: object
    required: [invoice_id, context]
    properties:
      invoice_id: { type: string, description: "Invoice number" }
      context: { type: string }
  output:
    type: object
    properties:
      summary: { type: string }
      vendor_name: { type: string }
      total_amount: { type: number }
```

Option B – Python Classes

```yaml
io_schema:
  input_class: app.schemas.InvoiceInput
  output_class: app.schemas.InvoiceOutput
```

AgentDock dynamically imports these classes and validates them using Pydantic.

5.2.5 Arguments & Runtime Config

Key-value pairs used to control execution or environment.
AgentDock merges these into runtime context passed to the adapter.

```yaml
arguments:
  temperature: 0.3
  max_retries: 3
  timeout_sec: 30
```

5.2.6 Policies Block

Defines security and governance rules.

```yaml
policies:
  tools:
    allowed: [bm25_search, dense_search]
    deny_by_default: true
  safety:
    redact_patterns:
      - "\b\d{3}-\d{2}-\d{4}\b"   # SSN
      - "(?i)password"
    max_output_chars: 3000
    halt_on_violation: true
```

Behavior

- Input redaction runs *before* agent call.
- Output redaction runs *after* agent call.
- Violations trigger `PolicyViolationError (403)` if `halt_on_violation = true`.

5.2.7 Auth Block

Authentication & authorization configuration.

```yaml
auth:
  mode: jwt
  jwt:
    secret_key: "${JWT_SECRET}"
    expiry_minutes: 120
```

```yaml
  api_keys:
    enabled: true
    rotation_days: 30
  roles:
    - name: admin
      permissions: [deploy, rollback, key_manage]
    - name: developer
      permissions: [invoke, view_metrics]
    - name: viewer
      permissions: [read_only]
```

- **Modes:** `jwt`, `oauth2`, or `none`
- JWT secrets and expiry required if `mode: jwt`.
- API keys auto-generated per role.
- All protected endpoints validate either header:
  - `Authorization: Bearer <token>` (JWT/OAuth2)
  - `x-api-key: <key>` (API Key auth)

5.2.8 Observability Block

Telemetry and logging configuration.

```yaml
observability:
  langfuse:
    project: "invoice-copilot"
    public_key: "${LANGFUSE_PUBLIC}"
    secret_key: "${LANGFUSE_SECRET}"
  tracing: true
  log_level: info
  metrics:
    latency: true
    tokens: true
    cost: true
```

- **LangFuse Integration:** optional but recommended; must specify keys.
- **Tracing:** enables OpenTelemetry spans for each node/tool.
- **Log level:** `debug | info | warn | error`.

5.2.9 Expose Block

Defines how the agent is exposed to the outside world.

```yaml
expose:
  rest: true
  streaming: sse      # Options: sse | websocket | none
  port: 8080
  host: 0.0.0.0
  cors:
    origins: ["*"]
    methods: ["GET", "POST"]
```

- Generates REST routes `/invoke`, `/stream`, `/schema`, `/health`.
- Enables CORS automatically for declared origins.

5.2.10 Schema Validation Rules

Rule	Enforcement
`version` is required	Must equal `"1.0"` for v1 parser
`agent.entrypoint`	Must resolve to callable
`model.provider`	Must be in supported provider list
`io_schema`	Must have valid input/output objects or importable classes
`auth`	Must contain at least one enabled mode
`observability`	Optional; warn if telemetry keys missing
`expose`	At least one of `rest` or `streaming` = true
`arguments`, `policies`	Optional, but validated if present

Errors are surfaced with path hints (e.g., `policies.safety.redact_patterns[1] invalid regex`).

5.2.11 Example Minimal Dockfile

```yaml
version: "1.0"
agent:
  name: doc_summarizer
  entrypoint: src.agent:build_graph
  framework: langgraph
model:
  provider: openai
  name: gpt-4o-mini
io_schema:
```

```
  input:
    type: object
    properties:
      document: { type: string }
  output:
    type: object
    properties:
      summary: { type: string }
auth:
  mode: api_key
  api_keys:
    enabled: true
expose:
  rest: true
```

This file alone should deploy a working FastAPI runtime with a `/invoke` endpoint protected by an API key.

5.2.12 Validation Lifecycle

1. **CLI → Parser:** YAML → Dict.
2. **Schema Validation:** Against Dockfile v1 JSON Schema.
3. **Entrypoint Validation:** Module import & callable check.
4. **Adapter Validation:** Confirms supported framework.
5. **Auth/Policy Validation:** Ensures proper structure.
6. **Final Pydantic model instantiation:** yields `DockSpec` object used by Builder.

5.2.13 Error Codes (Partial List)

Code	Description
`4001`	Invalid YAML structure
`4002`	Missing required fields
`4003`	Entrypoint import error
`4004`	I/O schema validation failed
`4005`	Auth configuration invalid
`4006`	Policy syntax error
`4007`	Unsupported framework
`4008`	Telemetry credentials missing when required

✅ Completion Criteria

The Dockfile specification is satisfied when:

- The file fully defines all required agent runtime metadata.
- AgentDock CLI can `validate` and `deploy` it without additional inputs.
- Generated API reflects declared schema and policies accurately.

5.3 Authentication & Authorization

5.3.1 Objectives

The authentication and authorization layer ensures that:

1. Only authenticated clients can access agent endpoints (`/invoke` , `/stream`).
2. Access privileges align with assigned **roles** and **permissions**.
3. Secrets and API keys are securely stored, rotated, and revocable.
4. All protected requests are traceable and auditable through logs and telemetry.

> 🎯 **Goal:** Every agent deployed through AgentDock should be secure out-of-the-box without developers writing any authentication or policy logic.

5.3.2 Supported Authentication Modes (v1)

AgentDock supports two modes in v1:

Mode	Description	Use Case
JWT (JSON Web Token)	Uses HMAC-signed tokens for stateless access.	Most enterprise/internal agents.
API Key	Simple key-based access for external or non-OAuth clients.	Public microservices or developer demos.

(OAuth2 planned for v2 with enterprise SSO integration.)

5.3.3 Authentication Flow

A. JWT Mode

1. Admin generates JWT secret key at deploy time (from Dockfile `auth.jwt.secret_key`).
2. When user logs in or requests a token:
   - Control Plane issues a signed JWT using HMAC SHA256.
   - Includes claims:

```
{
  "sub": "user@example.com",
  "role": "developer",
  "agent": "invoice_copilot",
  "exp": 1735843200
}
```

3. Client includes token in requests:

```
Authorization: Bearer <jwt_token>
```

4. Runtime middleware verifies:
   - Signature validity
   - Expiry
   - Role permissions (against Dockfile roles section)
5. Invalid or expired tokens → `401 Unauthorized` response.

B. API Key Mode

1. Admin generates key from Dashboard or CLI:

```
agentdock key create --agent invoice_copilot --role viewer
```

2. Key stored in hashed form in Postgres table `api_keys`.
3. Client includes header:

```
x-api-key: <api_key_value>
```

4. Runtime middleware:
   - Looks up hash in DB.
   - Confirms active status and role.
   - Logs usage in telemetry.

Rotation & Revocation

- Keys can be rotated via:

```
agentdock key rotate <agent> <key_id>
```

- Revocation sets `revoked_at` timestamp → immediate invalidation.

5.3.4 Authorization Model

Roles (Defined in Dockfile or Dashboard)

Role	Default Permissions	Notes
admin	deploy, rollback, view_metrics, manage_keys, invoke	Full control
developer	invoke, view_metrics, read_logs	Day-to-day users
viewer	view_metrics, read_docs	Read-only access

Roles map directly to permissions defined in Dockfile:

```
auth:
  roles:
    - name: admin
      permissions: [deploy, rollback, key_manage, invoke, view_metrics]
    - name: developer
      permissions: [invoke, view_metrics, read_logs]
```

Permission Enforcement

Each API endpoint in the runtime declares its required permission.

Example:

Endpoint	Required Permission
`/invoke`	invoke
`/stream`	invoke
`/metrics`	view_metrics
`/rollback`	rollback
`/docs`	read_docs

The **Auth Middleware** cross-checks the token's role → permissions list before execution.

5.3.5 Token & Key Storage

Secret Type	Storage Location	Security Control
JWT Secret Key	Encrypted at rest in Postgres (`secrets` table)	AES-256 encryption with service key
API Keys	Hashed (SHA256) in `api_keys` table	Compare hash at runtime
Role Mappings	`roles` table	Cached for low latency
Expiry Policies	Defined in Dockfile	Auto-expiry job (daily cron)

5.3.6 Middleware Design

The **Auth Middleware** runs before every request.

Pseudo Flow:

```
def auth_middleware(request):
    token = extract_token(request)
    claims = verify_token(token)
    role = claims.get("role")
    if not has_permission(role, request.endpoint):
        raise HTTPException(403, "Forbidden")
```

Middleware stack order:

```
[auth_middleware] → [policy_middleware] → [telemetry_middleware] →
[agent_adapter]
```

5.3.7 Rate Limits & Usage Quotas (MVP Light Version)

While v1 won't include complex metering, minimal protection is provided:

- **Static rate limits** configurable per role:

  ```
  auth:
    rate_limits:
      developer: 100 requests/minute
      viewer: 50 requests/minute
  ```

- Enforced via in-memory Redis counter (shared per runtime pod).

- Exceeding limit → `429 Too Many Requests`.

Metrics Logged

- Total requests per key/token per 5-min interval.
- Rate-limit breaches reported to LangFuse as “auth” events.

5.3.8 Dashboard Controls (v1)

Feature	Description
View Roles	See roles and permissions for an agent.
Generate Keys	Create/revoke API keys per role.
Token Expiry Monitor	Shows JWT expiry timelines.
Request Logs	View access attempts (IP, timestamp, endpoint).

Future v2:

- SSO & org-level role management
- Token analytics & cost correlation

5.3.9 Error Responses

Code	Error	Description
`401 Unauthorized`	Missing or invalid token	Signature invalid, token expired
`403 Forbidden`	Insufficient permissions	Role does not have required access
`429 Too Many Requests`	Rate limit exceeded	Retry after header included
`498 Token Expired`	Custom error for expired JWT	Refresh required
`499 Key Revoked`	API key manually revoked	Not accepted

All responses include structured error payload:

```
{
  "error": "Forbidden",
  "code": 403,
```

```
  "detail": "Role 'viewer' cannot invoke agent."
}
```

5.3.10 Acceptance Criteria

Requirement	Success Definition
JWT verification	Works with HMAC SHA256, rejects invalid/expired tokens
API key lookup	Constant-time hash match, revocation respected
Role-permission mapping	Enforced for all endpoints
Rate limits	Block requests > quota, log to LangFuse
Secure storage	Secrets encrypted and unreadable via DB queries
CLI integration	`agentdock key list` and `agentdock key revoke` functional

✅ Completion Definition

Authentication and authorization requirements are complete when:

- Every endpoint of an AgentDock runtime enforces **auth + role validation**.
- API keys and JWTs can be created, rotated, and revoked.
- Unauthorized calls are blocked and logged.
- No manual code is required for securing new agents.

Perfect — here's **Section 5.4 – Policy Engine** for your AgentDock v1 PRD.
This section formalizes the logic that protects agents from unsafe or unauthorized actions and ensures compliance during every invocation.

5.4 Policy Engine

5.4.1 Objective

The Policy Engine enforces **runtime governance** for all agents deployed through AgentDock.

It applies **tool access controls**, **input/output redaction**, and **safety validations** before and after each model invocation.

> 🎯 Goal: Guarantee that every agent call adheres to configured organizational rules without requiring custom validation code.

5.4.2 Execution Stages

Stage	Description	Trigger
Pre-Invoke	Validate request payload and redact sensitive information before the model or tool receives it.	Runs after Auth Middleware, before Adapter run.
In-Invoke	Enforce tool usage permissions during agent execution (for LangGraph nodes / tools).	Runs on every node/tool invoke event.
Post-Invoke	Inspect and optionally redact or truncate model output; verify compliance rules.	Runs after Adapter returns output, before response → client.

5.4.3 Policy Definition Format

Policies are declared inside the Dockfile → `policies` block.

```
policies:
  tools:
    allowed: [search_tool, summarize_tool]
    deny_by_default: true
  safety:
    redact_patterns:
      - "(?i)password"
      - "\b\d{3}-\d{2}-\d{4}\b"   # SSN
    max_output_chars: 4000
    block_prompt_injection: true
    halt_on_violation: true
```

Parsed Structure → Pydantic Model

```
class ToolPolicy(BaseModel):
    allowed: list[str]
    deny_by_default: bool = True
```

```python
class SafetyPolicy(BaseModel):
    redact_patterns: list[str] = []
    max_output_chars: int | None = None
    block_prompt_injection: bool = True
    halt_on_violation: bool = False
```

5.4.4 Pre-Invoke Processing

Actions

1. **Input Validation:** Ensure all required fields exist (already done by schema layer).
2. **PII Redaction:** Apply regex-based replacements on string fields.
3. **Prompt Injection Prevention:**
   - Detect `{“ignore previous instructions”}` patterns,
   - Strip suspicious control tokens (e.g., “system:”, “assistant:”).
4. **Policy Audit Log:** Log every applied redaction and match count.

Acceptance Criteria

- ✅ All regex patterns validated at deploy time.
- ✅ If `halt_on_violation = true`, matching sensitive patterns abort the run with `403 PolicyViolationError`.
- ✅ Redaction report included in LangFuse trace metadata.

5.4.5 In-Invoke Tool Gating

Purpose

Prevent unauthorized tool executions inside multi-tool graphs.

Mechanism

- The Adapter emits an event for every node/tool execution: `tool_invoked(name="bm25_search")`
- Policy Engine intercepts event stream; if `name` ∉ `allowed`, raise `ToolDeniedError`.

Acceptance Criteria

- ✅ Unauthorized tool → immediate 403 error.
- ✅ Allowed tool list loaded from Dockfile → cached in runtime.

- ✅ All tool invocations recorded in telemetry (tracked via LangFuse span tags).

5.4.6 Post-Invoke Processing

Actions

1. **Output Redaction:** Apply configured patterns to remove PII from generated text.
2. **Output Length Guard:** Truncate responses exceeding `max_output_chars`.
3. **Compliance Check (Experimental):** Scan for flagged keywords (“confidential”, “restricted”).
4. **Violation Handling:** If `halt_on_violation=True`, raise `PolicyViolationError`.

Example Violation Response

```
{
  "error": "PolicyViolationError",
  "code": 403,
  "detail": "Output contained forbidden term 'confidential'.",
  "policy": "safety.redact_patterns"
}
```

5.4.7 Policy Evaluation Flow

5.4.8 Policy Violation Handling

Violation Type	Error Code	Action
PII Detected	403	Redact / abort based on halt flag
Unauthorized Tool	403	Immediate abort of agent run
Output Too Long	413	Truncate / warn
Regex Compile Error	400	Validation error during deploy

All violations generate a structured telemetry event tagged `policy_violation=True`.

5.4.9 Logging & Telemetry

- Every PolicyEngine action produces a LangFuse event:

```
{
  "type": "policy_event",
  "stage": "pre-invoke",
  "matches": 3,
  "policy": "safety.redact_patterns",
  "duration_ms": 5
}
```

- Violations are highlighted in the dashboard with agent/version context.
- Aggregated metrics: violations per 100 runs, redacted fields count, blocked tool calls.

5.4.10 Acceptance Criteria

Requirement	Success Definition
Pre-invoke redaction	Sensitive inputs masked before model call
Tool allow/deny	Unauthorized tools blocked
Output filters	Redactions/truncations applied post-run
Violations observable	Logged + visible in dashboard
Performance	Policy check adds < 10 ms overhead per call
Config validation	Regex & policy syntax validated at deploy time

---

✅ Completion Definition

The Policy Engine is complete when:

- Every agent invocation executes through the Pre-Invoke → In-Invoke → Post-Invoke pipeline.
- All redaction and authorization policies are configurable via Dockfile.
- Violations are logged, traced, and optionally halt execution.
- Developers need **no manual code** for compliance logic.

5.5 Telemetry & Monitoring

5.5.1 Objective

AgentDock v1 integrates a unified telemetry and monitoring layer so that every deployed agent is *observable by default*.
The purpose is to give developers immediate insight into:

- Latency and throughput
- Token usage and LLM costs
- Success / failure rates
- Tool-level and node-level timings
- Policy and auth events

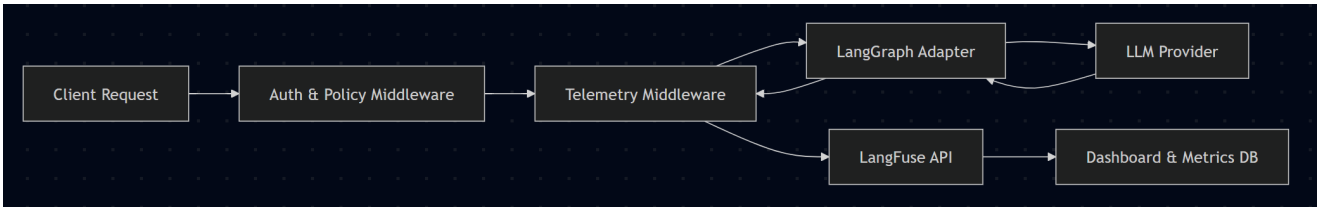
> 🎯 **Goal:** Enable teams to measure, debug, and optimize agents without adding any manual logging or metrics code.

5.5.2 Telemetry Architecture

Components

Component	Description
LangFuse Integration	Primary telemetry backend; receives structured traces, spans, and metrics.
Local Metrics Collector	Lightweight fallback that aggregates metrics when LangFuse is unreachable.
Telemetry Middleware	FastAPI middleware that wraps each request and emits metrics.
Event Emitter	Emits `run_start`, `run_end`, `policy_event`, `error_event`, and `tool_event` messages.
Dashboard Visualization	Consumes LangFuse data + internal metadata for charts and health cards.

Data Flow



5.5.3 Captured Metrics

A. Request-Level

Metric	Description	Source
`latency_ms`	End-to-end time from request start to response	Middleware timer
`status_code`	HTTP status	FastAPI
`tokens_input` / `tokens_output`	Token counts from model response	Adapter
`cost_usd`	Approx USD cost computed from provider rates	Adapter
`success` / `error_type`	Result classification	Runtime
`user_role`	From JWT/API Key	Auth Middleware

B. Tool / Node Level

Metric	Description
`tool_name` + `duration_ms`	Time per tool/node
`tool_error`	Captures failures per node
`tool_call_count`	Number of invocations

C. Policy Events

- Redaction matches, violation counts, blocked tools.

D. Resource Metrics

- Container CPU %, Memory %, Request Rate (req/s).
- Logged locally and exposed at `/metrics` (Prometheus format).

5.5.4 LangFuse Integration

Flow

1. Runtime initializes LangFuse client using credentials from Dockfile.
2. On every `/invoke` or `/stream`:
   - `run_start` event logged.
   - Each tool → LangFuse span.
   - `run_end` includes latency, token counts, cost, error flags.
3. All traces tagged with:

- `agent_name`, `version`, `environment`, `user_id`.

Example Payload

```
{
  "run_id": "lf_9a4b...",
  "agent": "invoice_copilot",
  "version": "v1.2",
  "latency_ms": 2140,
  "tokens_input": 1124,
  "tokens_output": 731,
  "cost_usd": 0.0123,
  "success": true
}
```

Acceptance Criteria

- ✅ Every request produces a LangFuse trace ID.
- ✅ Metrics visible in LangFuse UI within 5 seconds.
- ✅ Missing telemetry → recorded to local logs for backfill.

5.5.5 Local Metrics Collector (Fallback)

- Aggregates metrics in SQLite / CSV when LangFuse is disabled.
- Periodically flushes to LangFuse once connectivity returns.
- CLI support:

```
agentdock metrics export --agent invoice_copilot
```

5.5.6 Dashboard Display Requirements

View	Key Elements
Agent Summary	Name, version, uptime, requests / hour, success %, avg latency
Latency Graph	p50/p90/p99 charts
Token Usage Chart	Tokens IN/OUT per hour/day
Cost Tracker	Estimated $ per 1000 calls
Error Breakdown	Counts by error type (Auth, Policy, Runtime)
Tool Stats	Time spent per tool node
Policy Events Panel	Violations and redactions count

View	Key Elements
LangFuse Link	Deep link to trace viewer

5.5.7 Metric Emission Standards

Category	Format	Sampling
Latency	`latency_ms` (int)	100 % of requests
Tokens / Cost	float	100 %
Logs / Traces	JSON lines	100 %
Resource Metrics	Prometheus text	15 s interval

All telemetry messages follow this envelope:

```
{
  "event": "run_end",
  "timestamp": "2025-11-02T10:12:55Z",
  "agent": "invoice_copilot",
  "data": {...}
}
```

5.5.8 Error Handling & Resilience

- If LangFuse API unreachable → auto retry 3× then store offline.
- Telemetry failure ≠ API failure; agent still responds.
- Local telemetry retention = 7 days (default).

5.5.9 Performance Targets

Metric	Target
Telemetry overhead	≤ 5 % added latency
Trace delivery time	≤ 3 s after run end
Storage retention	30 days in LangFuse (default)
Data loss rate	< 0.1 % per 10 k requests

5.5.10 Acceptance Criteria

Requirement	Success Definition
LangFuse integration enabled	Traces visible for all runs
Local metrics fallback	Works offline and syncs when re-connected
Dashboard latency / token graphs	Render accurately from metrics DB
Policy / Auth events correlated	Trace links visible in LangFuse
Prometheus / metrics endpoint	`/metrics` returns valid format
Performance budget met	Overhead < 5 % p95 latency

✅ Completion Definition

Telemetry & Monitoring module is complete when:

- Every agent invocation automatically emits LangFuse traces and local metrics.
- Developers can visualize latency, cost, and policy events without extra code.
- Failures never affect the agent's response path.

Perfect — here's the complete **Section 5.6 – API Exposure & Schema Validation** of your AgentDock v1 PRD.
This section describes how the system auto-generates APIs for every deployed agent and enforces strict schema validation across all inputs and outputs.

5.6 API Exposure & Schema Validation

5.6.1 Objective

Every agent deployed through AgentDock must automatically expose a **standardized API interface**—secure, schema-validated, and fully documented via OpenAPI 3.0.

> 🎯 **Goal:** Developers should be able to call any deployed agent via `/invoke` or `/stream` endpoint with zero manual code, knowing that all input/output conforms to the Dockfile-defined schema.

5.6.2 Core Design Principles

1. **Consistency:** Every agent exposes identical endpoint patterns.
2. **Validation:** All request/response payloads validated by Pydantic models generated from Dockfile.
3. **Discoverability:** Auto-generated Swagger & OpenAPI specs for immediate inspection.
4. **Interoperability:** APIs conform to REST and streaming standards (SSE / WebSocket).
5. **Security:** All endpoints protected by JWT/API Key middleware.

5.6.3 Endpoints Overview

Endpoint	Method	Description	Auth Required	Streaming
`/invoke`	POST	Invoke the agent synchronously with validated input	✅	❌
`/stream`	GET	Stream token or event-level responses	✅	✅
`/schema`	GET	Retrieve agent's input/output schema in JSON Schema form	❌	❌
`/health`	GET	Health probe and version info	❌	❌
`/metrics`	GET	Prometheus-format metrics endpoint	✅	❌

5.6.4 Endpoint Definitions

A. /invoke

Request

```
POST /invoke
Authorization: Bearer <jwt>
Content-Type: application/json
```

Example Payload (based on Dockfile schema):

```
{
  "invoice_id": "INV-1234",
```

```
  "context": "Q2 2025 billing statement"
}
```

Response

```
{
  "summary": "Invoice INV-1234 from Acme Corp for $432.50 on 2 May 2025",
  "vendor_name": "Acme Corp",
  "total_amount": 432.5
}
```

Behavior

- Parses body using Pydantic model from Dockfile I/O schema.
- Executes pre-invoke → agent → post-invoke pipeline.
- Emits telemetry events for request latency, cost, and outcome.

B. /stream

Supports real-time token or event streaming via SSE or WebSocket.

Example (SSE)

```
curl -N -H "Accept: text/event-stream" -H "Authorization: Bearer <token>" \
  http://localhost:8080/stream?input='{"invoice_id":"INV-1234"}'
```

Event Stream:

```
data: {"event":"token","content":"Invoice"}
data: {"event":"token","content":"total"}
data: {"event":"completed"}
```

Behavior

- Streaming adapter uses async generator.
- Emits `start`, `token`, `completed`, and `error` events.
- Tied into LangFuse trace spans for event-level latency.

C. /schema

Returns validated JSON Schema for input and output.

```
GET /schema
```

Response:

```
{
  "input_schema": {
    "type": "object",
    "properties": {
      "invoice_id": {"type": "string"},
      "context": {"type": "string"}
    }
  },
  "output_schema": {
    "type": "object",
    "properties": {
      "summary": {"type": "string"},
      "total_amount": {"type": "number"}
    }
  }
}
```

D. /health

Lightweight endpoint used for readiness/liveness probes.

Response:

```
{
  "status": "healthy",
  "version": "1.0.3",
  "agent": "invoice_copilot",
  "uptime_seconds": 86400
}
```

E. /metrics

Exposes Prometheus-style metrics for infra monitoring.

Response example:

```
agentdock_requests_total{agent="invoice_copilot"} 1032
agentdock_latency_avg_ms{agent="invoice_copilot"} 182.4
```

5.6.5 Schema Generation Process

1. Parse I/O definitions from Dockfile (either inline or via Python imports).
2. Convert them into Pydantic models dynamically:

```
InputModel = create_model("InvoiceInput", **fields_from_dockfile("input"))
OutputModel = create_model("InvoiceOutput", **fields_from_dockfile("output"))
```

3. Auto-generate OpenAPI specification from FastAPI router:

```
app = FastAPI()
app.include_router(agent_router)
openapi_schema = app.openapi()
```

4. Serve `/docs` (Swagger UI) and `/openapi.json` endpoints automatically.

5.6.6 Schema Validation Rules

Check	Condition	Error
Missing required fields	Input model fields not provided	`422 ValidationError`
Invalid type	Type mismatch vs schema	`422 ValidationError`
Output violation	Agent response doesn't match OutputModel	`500 SchemaMismatchError`
Missing schema	Dockfile missing I/O block	`400 InvalidDockfileError`

Example Error Response

```
{
  "error": "ValidationError",
  "code": 422,
  "details": [
    {"field": "invoice_id", "msg": "field required"}
  ]
}
```

5.6.7 OpenAPI Documentation

Every deployed runtime automatically generates an OpenAPI 3.0 spec.

Artifact	URL
JSON Spec	`/openapi.json`
Swagger UI	`/docs`
ReDoc UI	`/redoc`

Spec Enrichment:

- Auto-includes input/output schemas.
- Adds `securitySchemes` (JWT + API Key).
- Tags endpoints under `AgentDock Runtime`.

5.6.8 Streaming Protocol Specification

SSE (Server-Sent Events)

- Content-Type: `text/event-stream`
- Events: `start`, `token`, `end`, `error`
- Each event: `data: { "event": "...", "content": "..." }`

WebSocket (optional)

- Endpoint: `/ws/stream`
- Message types:
  - `{ "type": "start" }`
  - `{ "type": "token", "data": "..." }`
  - `{ "type": "end" }`

Both modes support reconnect and heartbeat.

5.6.9 Security Considerations

Endpoint	Protection	Notes
`/invoke`, `/stream`	JWT/API key middleware	Mandatory
`/schema`, `/health`	Public	For discovery and health probes

Endpoint	Protection	Notes
`/metrics`	JWT (admin role only)	Sensitive telemetry
Rate limiting	Applied per token/key	Enforced globally

5.6.10 Acceptance Criteria

Requirement	Success Definition
`/invoke` endpoint functional	Returns valid response using schema
`/stream` endpoint streams tokenized output	SSE/WebSocket both supported
Schema validation	Rejects invalid input; validates output
OpenAPI spec	Available at `/openapi.json` with correct metadata
Error structure	Consistent JSON error schema
Security	Auth required for all stateful endpoints

✅ Completion Definition

API Exposure & Schema Validation is complete when:

- Any Dockfile-deployed agent automatically exposes `/invoke`, `/stream`, `/schema`, `/health`, and `/metrics`.
- Input/output schemas are dynamically validated and documented.
- Auth and policy layers protect all stateful routes.
- The OpenAPI schema fully represents the deployed agent.

Excellent — here's the **Section 5.7 – Deployment & Runtime Management** of your AgentDock v1 PRD.
This section defines how each agent is packaged, deployed, versioned, monitored, and maintained at runtime — forming the **DevOps backbone** of the entire platform.

5.7 Deployment & Runtime Management

5.7.1 Objective

To manage the **complete lifecycle** of a deployed agent—from build → launch → monitoring → rollback—through automated, repeatable, and secure runtime orchestration.

> 🎯 **Goal:** Enable one-command deployments (`agentdock deploy`) that result in a fully functional, isolated runtime environment for each agent with version tracking, rollback, and health monitoring.

5.7.2 Core Architecture

AgentDock deployment workflow consists of **three coordinated components**:

Component	Function
Builder Service	Converts Dockfile → container image with FastAPI runtime.
Controller Service	Orchestrates runtime lifecycle (create, update, rollback, delete).
Runtime Agent	The actual deployed container/pod running the agent code.

Each deployment emits build logs, creates metadata records, and registers telemetry hooks.

5.7.3 Build Process

Step-by-Step Flow

1. Validate Dockfile (Section 5.2).
2. Generate runtime boilerplate:
   - `main.py` (FastAPI entrypoint)
   - Auth/policy middleware code
   - Dockerfile (base: `python:3.11-slim`)
3. Create a temporary workspace:
   - Copy agent source directory.
   - Add dependencies (`requirements.txt`).
4. Build image:

```
docker build -t agentdock/invoice_copilot:v1.0 .
```

5. Push image (optional):

```
docker push agentdock/invoice_copilot:v1.0
```

6. Controller launches container/pod.

Build Specifications

Parameter	Default	Description
Base image	`python:3.11-slim`	Minimal runtime
Ports exposed	`8080`	REST entrypoint
Startup script	`/app/run.sh`	Runs `uvicorn main:app`
Environment vars	Derived from Dockfile	Injected at runtime
Secrets	Mounted via secure volume	Never hardcoded

Acceptance Criteria

- ✅ Image built reproducibly with content hash.
- ✅ No plain secrets in Docker layers.
- ✅ Build completes < 2 min for average agent.

5.7.4 Deployment Strategies

A. Local (Developer Mode)

- Uses **Docker Compose** to deploy single or multiple local agents.
- Ideal for debugging or rapid prototyping.
- Stored in `~/.agentdock/local_registry.json`.

B. Enterprise (Cluster Mode)

- Uses **Kubernetes** as runtime orchestrator.
- Deployments managed via `AgentDockController` custom resource.
- Includes:
  - Replica management
  - Health probes
  - ConfigMap + Secret mounts
  - Autoscaling hooks

Example CRD Manifest:

```yaml
apiVersion: agentdock.io/v1
kind: AgentRuntime
metadata:
  name: invoice-copilot
spec:
  image: agentdock/invoice_copilot:v1.0
  replicas: 2
  port: 8080
  envFrom:
    - secretRef: jwt-secret
  resources:
    requests:
      cpu: "500m"
      memory: "512Mi"
```

5.7.5 Runtime Management

Controller Responsibilities

Operation	Description
Create	Deploy container/pod and register metadata
Update	Trigger new version deployment
Rollback	Revert to previous stable image
Delete	Gracefully terminate agent
Monitor	Continuously check health, latency, error rates

Controller exposes internal API for the Dashboard:

```
/controller/agents
/controller/deploy
/controller/rollback
/controller/status
```

Health Checks

Every runtime must expose:

- `/health` → returns `{status, uptime, version}`
- Kubernetes probes:

```
livenessProbe:
  httpGet:
    path: /health
    port: 8080
  initialDelaySeconds: 5
readinessProbe:
  httpGet:
    path: /health
    port: 8080
  periodSeconds: 10
```

Failure Handling

Scenario	Controller Action
Startup fails	Roll back to last stable version
Health check fails repeatedly	Mark as “degraded”
CrashLoop detected	Restart pod (max 3 retries)
Version not healthy after 2 min	Auto-rollback

5.7.6 Versioning & Rollback

Each deployment creates an immutable version record in Postgres:

Field	Description
`version`	Semantic version string (v1.0.2)
`image_ref`	Docker image tag
`status`	active / inactive / degraded
`created_at`	Timestamp
`deployed_by`	User who triggered deploy
`dockfile_yaml`	Full Dockfile snapshot

Rollback flow:

```
agentdock rollback invoice_copilot v1.0.1
```

→ Controller stops v1.0.2 → redeploys v1.0.1 image → validates health.

Acceptance Criteria

- ✅ Versioning metadata immutable.
- ✅ Rollback completes < 30 s.
- ✅ Telemetry retained across versions.

5.7.7 Logging & Observability

All runtimes forward logs to a centralized aggregator (FluentBit / Loki).

Log Format

```
{
  "timestamp": "2025-11-02T19:45:10Z",
  "agent": "invoice_copilot",
  "version": "v1.0.0",
  "level": "INFO",
  "message": "Agent invoked successfully",
  "latency_ms": 2010
}
```

Logs are correlated with LangFuse traces using `trace_id`.

5.7.8 Environment Management

Type	Handling	Description
Secrets	Mounted via Kubernetes Secrets or .env vault	For JWT, API keys
Configs	Derived from Dockfile and CLI args	Environment variables
Resources	CPU/memory requests defined per agent	Enforced by Controller

Example runtime injection:

```
ENV MODEL_NAME=gpt-4o
ENV TIMEOUT_SEC=20
```

5.7.9 Deployment CLI Commands

Command	Purpose
`agentdock deploy`	Validate → build → deploy new version
`agentdock validate`	Dry-run schema & policy validation
`agentdock rollback <agent> <version>`	Roll back to previous version
`agentdock logs <agent>`	Tail container logs
`agentdock status <agent>`	View deployment health/status
`agentdock cleanup`	Remove old builds and cache

All commands support `--verbose` and `--json` flags for automation pipelines.

5.7.10 Monitoring & Alerting

- Integrates with Prometheus (scrapes `/metrics`).
- Alert rules configurable per agent:

```yaml
alerts:
  - name: HighLatency
    condition: latency_p95 > 2000
    severity: warning
  - name: FailureRate
    condition: error_rate > 0.05
    severity: critical
```

- Sends alerts via Webhook / Slack / PagerDuty.

5.7.11 Performance Targets

Metric	Target
Average deploy time	≤ 90 s
Rollback time	≤ 30 s
Cold start latency	≤ 2 s
Runtime uptime	≥ 99.5 %
Controller reconciliation loop	≤ 10 s

5.7.12 Acceptance Criteria

Requirement	Success Definition
Build → deploy automation	Single CLI command fully deploys
Rollback	Immediate revert to previous version
Health monitoring	Automatic degradation detection
Versioning	Metadata persisted and queryable
Logging	Centralized and correlated with telemetry
Security	Secrets isolated and never stored in logs

✅ Completion Definition

Deployment & Runtime Management module is complete when:

- Each Dockfile can be transformed into a live containerized agent with health monitoring, versioning, and rollback.
- Builds, deployments, and recoveries can be managed entirely via the AgentDock CLI or Controller API.
- No manual DevOps intervention is needed for day-to-day agent lifecycle operations.

Excellent — here's the **Section 5.8 – Dashboard & Developer Console** of your AgentDock v1 PRD.
This section defines the **frontend and UX layer** of the platform that connects all modules — deployment, telemetry, auth, and policy — into a unified control interface for developers and admins.

5.8 Dashboard & Developer Console

5.8.1 Objective

The **AgentDock Dashboard** is a centralized web interface that allows developers, operators, and admins to:

- Manage agents (deploy, rollback, pause, delete)
- View live metrics, logs, and costs
- Inspect Dockfiles, schemas, and policies

- Generate and manage API keys or tokens
- Explore LangFuse traces and request histories

> 🎯 **Goal:** Provide a “mission-control” experience for LLM agent operations — similar to a hybrid of LangFuse + Kubernetes Dashboard + Postman, but tailored to the AgentDock ecosystem.

5.8.2 High-Level Architecture

Layer	Technology	Description
Frontend UI	React + Vite + Tailwind + ShadCN UI	Modern, responsive dashboard
API Gateway	FastAPI / GraphQL (BFF)	Communicates with Controller, Metrics DB, and LangFuse
Data Stores	Postgres + LangFuse API	Agent metadata, metrics, and audit logs
Auth	JWT / API Key Middleware	Same authentication mechanism as runtime
Deployment Events	WebSocket / SSE	Real-time updates for builds, metrics, and logs

---

5.8.3 Primary Views

A. Dashboard Home

Summary snapshot of all deployed agents.

Section	Description
Agent List Table	Name, status (active/degraded), version, uptime
Quick Actions	Deploy / Rollback / Delete
Health Cards	p95 latency, request rate, error rate
Cost Overview	Estimated $ spent per agent from LangFuse
System Status Banner	Alerts for controller or telemetry failures

---

B. Agent Details Page

Tabs for each deployed agent:

1. **Overview** – summary, version, maintainer metadata
2. **Metrics** – charts for latency, tokens, cost, violations
3. **Deployments** – history table (version, deployed by, timestamp)
4. **Logs** – live stream and filters (INFO/WARN/ERROR)
5. **Policies** – active policy configuration + violations trend
6. **Schema** – input/output schemas (JSON view + copy button)
7. **OpenAPI** – interactive API tester (Swagger embed or Postman-style UI)
8. **Keys & Auth** – view/create/revoke API keys per role
9. **LangFuse Link** – deep link to trace viewer for agent

C. Deployment View

Feature	Description
Build Logs Viewer	Tail real-time logs from AgentDock Controller
Deploy Progress Bar	Stepwise status (validating → building → pushing → starting → healthy)
Version Selector	Dropdown to view previous builds and roll back
Dockfile Preview	Read-only viewer for YAML spec used in deployment

---

D. Key Management View

- Generate, rotate, and revoke API keys.
- Each key associated with a role (`admin` , `developer` , `viewer`).
- Download as JSON or copy to clipboard.
- Expiry and last-used timestamps visible.
- Supports bulk revocation and regeneration.

E. Monitoring & Alerts View

Chart / Panel	Description
Latency & Error Rates	Aggregated over time using Prometheus metrics
Token Usage	Token IN/OUT vs. Cost per hour

Chart / Panel	Description
Policy Violations	Redaction or tool-block counts
Auth Events	Successful vs. failed logins
Custom Alerts	User-defined conditions (latency, failure rate, cost)

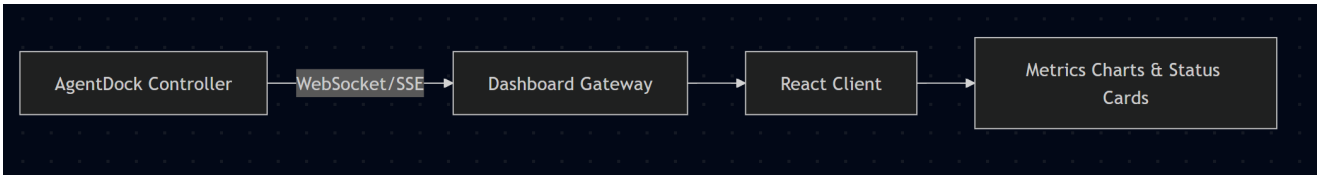
F. System Admin Panel

For platform admins only:

- Controller node status (online/offline)
- Cluster health summary (Kubernetes pods)
- LangFuse API status + token validity check
- Storage usage (Postgres, logs, cache)
- Configurable retention policies for telemetry and logs

5.8.4 Real-Time Data Architecture

The dashboard uses event-driven updates:



- Backend gateway maintains persistent WebSocket connections for each agent.
- UI subscribes to `build_event`, `health_event`, `metric_event` topics.

5.8.5 User Roles and Access

Role	Access Scope
Admin	Full access to deployments, logs, keys, metrics, alerts
Developer	Can deploy and view logs & metrics, no key management
Viewer	Read-only access to metrics, docs, schema
Anonymous	Can view health & schema only (if enabled)

All views enforce Auth middleware consistent with runtime endpoints.

5.8.6 UI Design Principles

1. **Minimal and Data-Dense:** Dashboard prioritizes metrics over decorative UI.
2. **Dark Mode First:** Easier on developer eyes during monitoring sessions.
3. **Responsive Design:** Works across desktop, tablet, and terminal views.
4. **Interactive Playground:** Users can test agent API from the browser with auth.
5. **Auditability:** Each action (logs, deploy, key rotate) is logged in Postgres (`audit_log` table).

5.8.7 Integration with Backend Services

Service	Integration
Controller API	Deployments, rollbacks, status updates
LangFuse API	Traces and metrics visualization
Auth Service	JWT validation + role-based UI permissions
Prometheus	Metrics data for charts (latency, requests rate)
Postgres	Persisted metadata and user actions history

5.8.8 Developer Console Tools

The Dashboard includes a **Developer Console section**, providing:

- Command palette (`Ctrl + K`) for common actions (deploy, logs, keys).
- Terminal-style CLI viewer to run commands in-browser.
- “YAML Builder” to generate Dockfiles visually (drop-down fields mapped to Dockfile schema).
- Real-time “Schema Validator” — paste Dockfile and get validation results live.

5.8.9 Telemetry & Analytics on Dashboard Use

All UI events (e.g., deploy triggered, log view opened) are captured via internal telemetry for UX analytics.
This data is stored in a separate `ui_events` table for usage analysis and future recommendations.

5.8.10 Performance Targets

Metric	Target
Page Load Time	≤ 2 s for main dashboard view
Real-Time Update Latency	≤ 1 s (WebSocket RTT)
Concurrent Users (per org)	100 + without lag
Chart Render FPS	≥ 30 FPS with 100 data points
Build Log Stream Delay	≤ 500 ms from controller event

5.8.11 Acceptance Criteria

Requirement	Success Definition
Agent table and metrics visible	Agents appear with real-time health and cost data
Deploy/Rollback actions functional	Works via Controller API with status feedback
Logs stream live	Real-time log tail for each agent
Key management secure	Create/revoke with role mapping
Policies visible	Policy violations and rules inspectable in UI
Auth integration	Role-based UI restriction active
OpenAPI docs viewable	Embedded Swagger viewer loads runtime spec
Dashboard responsive	Layout scales properly across devices

✅ Completion Definition

The Dashboard & Developer Console is complete when:

- All functional modules (deployments, metrics, logs, keys, schemas) are accessible through a single web interface.
- Real-time updates and role-based permissions work end-to-end.
- Developers can fully operate agents without touching CLI or Kubernetes directly.
- The UI remains lightweight, responsive, and secure by default.

---

6. Technical Architecture

6.1 Objective

To describe the **system-wide architecture** of AgentDock v1 — including modules, communication flows, data stores, and deployment topology — enabling independent teams to build, deploy, and scale components consistently.

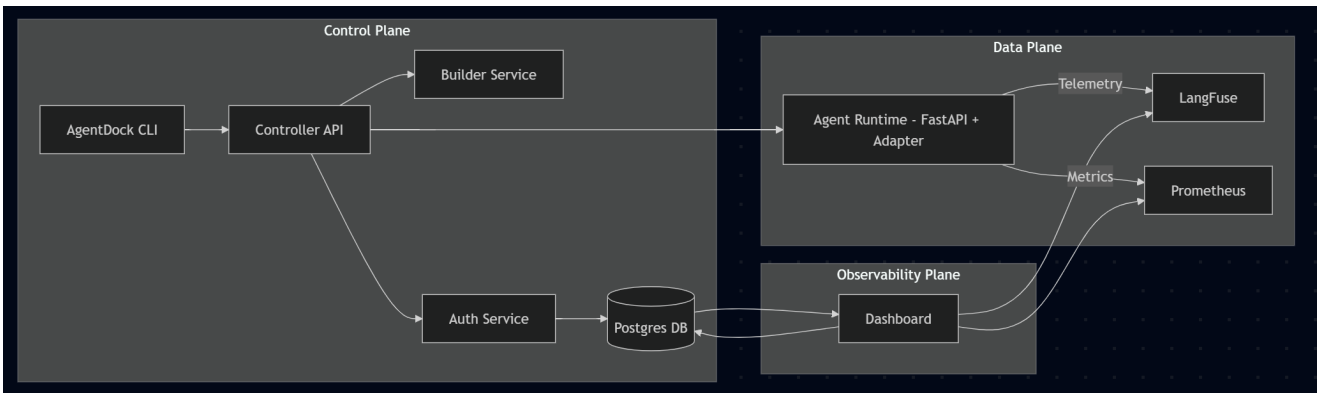
> 🎯 **Goal:** Achieve modularity, observability, and secure interoperability between the control plane, execution runtimes, and developer interfaces.

6.2 High-Level Overview

AgentDock follows a **three-plane design**:

Plane	Responsibility	Components
Control Plane	Manages deployments, configurations, and version metadata	Controller API, Builder Service, Auth Service
Data Plane	Executes agent runtimes and handles user traffic	Runtime Containers (FastAPI + LangGraph Adapters)
Observability Plane	Collects telemetry, logs, and metrics	LangFuse, Prometheus, Postgres, Dashboard Gateway

6.2.1 Macro View



6.3 Core Components

6.3.1 Controller Service

- **Type:** FastAPI microservice
- **Responsibilities:**
  - Receive deploy/rollback requests from CLI or Dashboard
  - Validate Dockfile and schema
  - Coordinate builds via Builder Service
  - Register versions and health in Postgres
  - Expose REST API for Dashboard integration
- **Endpoints:**
  - `/controller/deploy` – trigger deployment
  - `/controller/rollback` – revert version
  - `/controller/status` – return runtime health

6.3.2 Builder Service

- **Function:** Converts Dockfile into deployable container image.
- **Flow:**
  1. Parse Dockfile → validate.
  2. Generate runtime scaffold (main.py, Dockerfile).
  3. Build and tag image.
  4. Push to registry.
  5. Notify Controller via WebSocket/SSE.
- **Tech Stack:** Python, Docker SDK, GitPython, Redis Queue (for parallel builds).

6.3.3 Auth Service

- Provides JWT signing and API key management.
- Validates role claims for Dashboard and runtime APIs.
- Exposes gRPC and REST interfaces for low-latency checks.

6.3.4 Runtime Service

- Self-contained FastAPI application per agent.
- Auto-generated by Builder.

- Includes middlewares: Auth → Policy → Telemetry.
- Uses LangGraphAdapter for LLM calls and tool invocations.

6.3.5 Dashboard Gateway

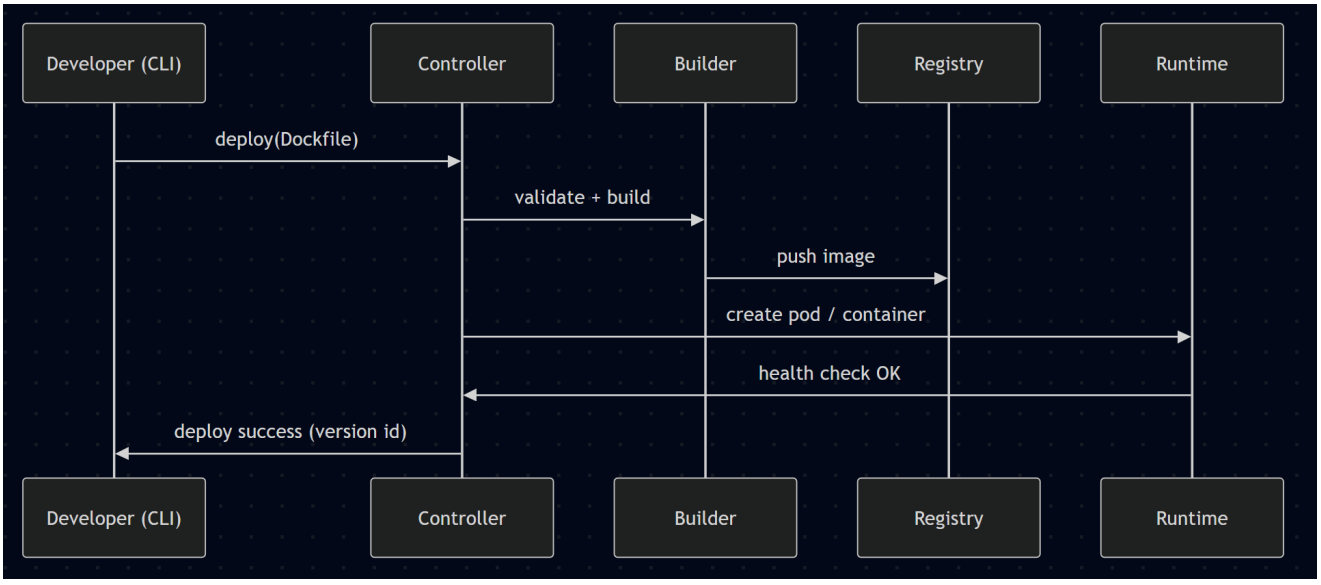
- **Purpose:** Bridge between UI (frontend) and backend services.
- **Acts as BFF (Backend-for-Frontend):**
  - Aggregates data from Controller, LangFuse, and Prometheus.
  - Provides GraphQL or REST API to React frontend.
  - Streams real-time events (WebSocket).

6.3.6 Database Layer

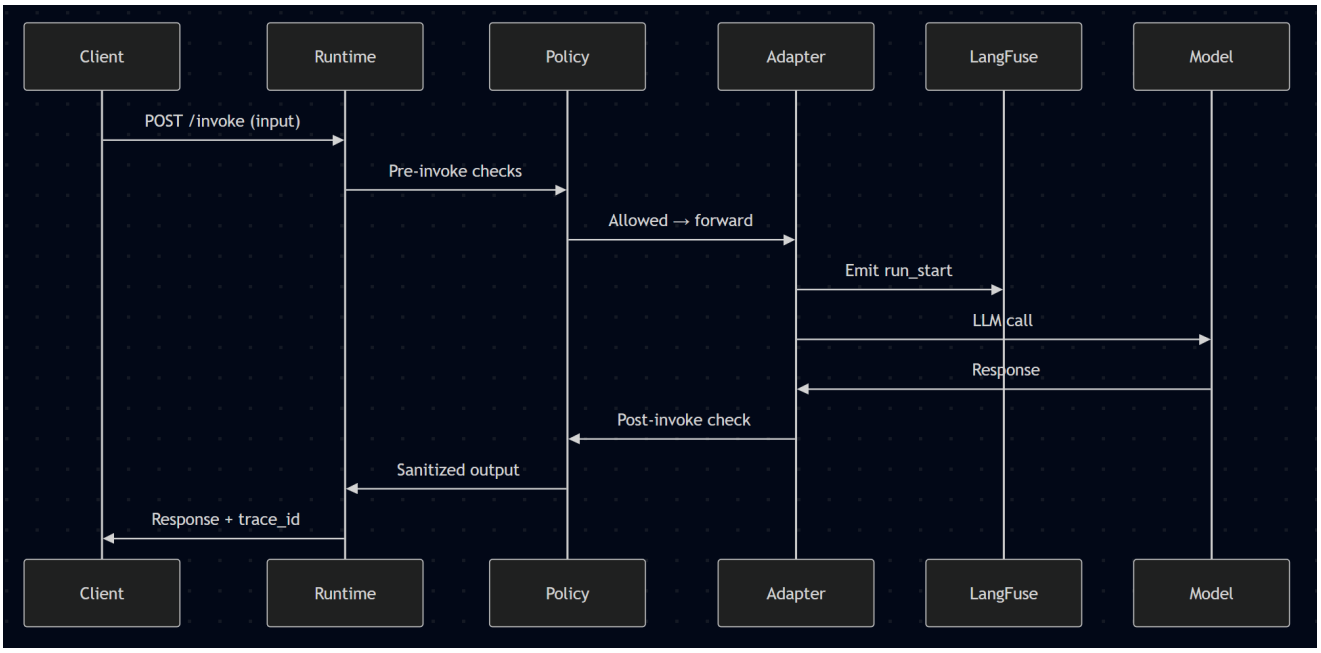
Database	Usage
Postgres	Core metadata – agents, versions, auth, audit logs
Redis	Build queues, rate-limit tracking, caching
Prometheus	Runtime metrics (through Exporter at port /metrics)
LangFuse Cloud	Trace storage and token analytics

6.4 Communication Flows

6.4.1 Deployment Flow



6.4.2 Invocation Flow



6.5 Internal APIs & Interfaces

6.5.1 Controller ↔ Builder

- Protocol: HTTP + WebSocket.
- Request:

```
{ "agent": "invoice_copilot", "dockfile_yaml": "...", "version": "1.0.2" }
```

- Response: Build status events (streamed).

6.5.2 Dashboard ↔ Controller

- REST/GraphQL queries for agent metadata.
- WebSocket stream for deployment status updates.

6.5.3 Runtime ↔ LangFuse

- Asynchronous API for trace creation and span reporting.

6.5.4 Runtime ↔ Prometheus

- Exposes `/metrics` endpoint scraped every 15 s.

6.5.5 Auth ↔ All

- JWT validation and role verification performed via shared public key (JWKS).

6.6 Security Architecture

Layer	Mechanism	Notes
Transport	HTTPS (TLS 1.3)	All internal/external comms
Authentication	JWT / API Key	Propagated through FastAPI middleware
Authorization	Role-based per endpoint	Evaluated at runtime
Secrets Management	Kubernetes Secrets / Vault	Mounted securely
Policy Enforcement	Pre-invoke & post-invoke middleware	Prevents unsafe operations
Audit Logging	Postgres `audit_log` table	For every admin action

6.7 Scalability Model

6.7.1 Horizontal Scaling

- Controller Service → Stateless → scale via K8s replicas.
- Builder Service → Parallel build queue (Redis).
- Runtime → Autoscale per agent (Deployment HPA rules).

6.7.2 Multi-Tenant Design

- Org ID tag on every resource (agent, build, trace).
- Role-based access segregation.
- Separate Postgres schemas per tenant (optional for v2).

6.7.3 High Availability

- PostgreSQL HA (2 replicas + async replica).
- Controller HA via Load Balancer + Sticky Sessions.
- Runtime pods auto-restarted on failure.

6.8 Deployment Topology

Environment	Description
Local Dev	Docker Compose with Controller, Builder, Runtime containers

Environment	Description
Staging	K8s namespace + Postgres + LangFuse sandbox
Prod	Managed K8s cluster (Azure AKS / GKE), Postgres HA, LangFuse Cloud, Prometheus Operator

---

6.9 Error Handling & Resilience

Layer	Failure	Mitigation
Builder	Image build failure	Retry ×3 → rollback → log error
Controller	DB connection loss	Fallback to read-replica
Runtime	Crash loop	Auto restart + graceful drain
Telemetry	LangFuse unreachable	Local buffer → retry
Auth	JWT key rotation	Use JWKS discovery endpoint

6.10 Technology Stack

Category	Tool / Framework
Core Backend	FastAPI (Python 3.11)
Runtime Adapter	LangGraph / LangChain SDK
Containerization	Docker + Kubernetes
Queue / Cache	Redis
Database	Postgres + SQLAlchemy
Observability	LangFuse, Prometheus, Loki
Frontend	React + Vite + Tailwind + ShadCN UI
Auth	JWT, OAuth2 (Planned v2)
CI/CD	GitHub Actions / CircleCI
Secrets	HashiCorp Vault / K8s Secrets

6.11 Performance Targets

Metric	Target
Avg Deployment Time	≤ 90 s
Avg Cold Start Latency	≤ 2 s
p95 Invoke Latency	≤ 1.8 s
Build Concurrency	20 parallel builds
API Uptime	≥ 99.5 %
Mean Recovery Time (MTTR)	< 60 s

6.12 Acceptance Criteria

Requirement	Success Definition
Services communicate via secure APIs	All internal calls over TLS + Auth
Controller ↔ Builder integration	Build events stream successfully
Runtime observability active	Telemetry + metrics visible in Dashboard
Autoscaling tested	Runtime replicas scale on load
Secrets isolated	No leakage in logs or images
CI/CD validated	Automated build + deploy pipeline green

✅ Completion Definition

The Technical Architecture is considered complete when:

- All core services (Controller, Builder, Runtime, Dashboard, Auth) have **defined interfaces and deployment patterns**.
- The system can be deployed on a **Kubernetes cluster** with full observability and HA.
- New teams can onboard using this document to implement extensions (e.g., multi-tenant v2) without architectural ambiguity.

---

7. System Components

7.1 Objective

Define the **internal design, data schema, and service contracts** for each major subsystem of AgentDock v1 so engineering teams can implement, extend, or debug individual modules confidently.

> 🎯 **Goal:** Ensure every component is self-contained, testable, and replaceable, with clearly defined interfaces and minimal coupling.

7.2 Module Summary

Module	Responsibility	Key Technologies
Controller Service	API gateway for deploy / rollback / status	FastAPI · Postgres · Redis
Builder Service	Dockfile→image conversion & runtime generation	Docker SDK · GitPython · Redis Queue
Auth Service	JWT & API-key lifecycle management	FastAPI · PyJWT · Postgres
Runtime Service	Per-agent FastAPI container executing graph	FastAPI · LangGraph Adapter
Policy Engine	Redaction & tool-gating middleware	Regex · Pydantic
Telemetry Agent	Metrics & trace emitter	LangFuse SDK · Prometheus Exporter
Dashboard Gateway	Aggregates data for frontend	GraphQL BFF · WebSockets
Frontend UI	Developer console	React · Tailwind · Vite
Database Layer	Persistence for metadata + logs	Postgres · SQLAlchemy
Queue Layer	Async build / notification bus	Redis · RQ Workers

---

7.3 Controller Service

7.3.1 Responsibilities

- Accept deployment / rollback / status requests.
- Persist version metadata in Postgres.
- Communicate with Builder Service asynchronously.
- Expose public API for Dashboard Gateway.

7.3.2 Core Classes

```
class AgentController:
    def deploy(self, dockfile: str, user: str) -> dict: ...
    def rollback(self, agent: str, version: str) -> dict: ...
    def status(self, agent: str) -> dict: ...
```

7.3.3 Database Models

Table	Columns
`agents`	id, name, description, framework, maintainer
`agent_versions`	id, agent_id, version, status, image_ref, created_at
`deployments`	id, version_id, runtime_url, health, deployed_by
`audit_log`	id, user, action, target, timestamp

7.3.4 Internal APIs

Method	Path	Purpose
POST	`/controller/deploy`	Trigger build + deploy
POST	`/controller/rollback`	Roll back agent
GET	`/controller/status/{agent}`	Health status
WS	`/controller/events`	Stream build updates

7.4 Builder Service

7.4.1 Pipeline Stages

1. **Parse Dockfile** → `DockSpec` object.
2. **Generate Scaffold** → `main.py`, `Dockerfile`.
3. **Build Image** → Docker SDK build().
4. **Push Image** → Registry (if configured).
5. **Notify Controller** → build events (WebSocket).

7.4.2 Queue Model

- Job enqueued to Redis as `BuildJob(id, dockfile_yaml, user)`
- Worker executes and updates job state in Postgres.

7.4.3 BuildJob States

State	Description
QUEUED	Waiting for worker slot
RUNNING	Image being built
FAILED	Error → Controller notified
COMPLETED	Image ready → deployed

7.5 Auth Service

7.5.1 Data Models

Table	Columns
`users`	id, email, password_hash (optional), role
`api_keys`	id, agent_id, key_hash, role, revoked_at
`roles`	id, name, permissions (JSONB)

7.5.2 API Endpoints

Method	Path	Description
POST	`/auth/token`	Issue JWT
POST	`/auth/verify`	Validate token
POST	`/auth/key/create`	Generate API key
DELETE	`/auth/key/revoke`	Revoke API key

7.5.3 Libraries

- PyJWT for signing
- bcrypt for key hashing
- AES-256 for secret encryption

7.6 Runtime Service

7.6.1 Internal Modules

Module	Role
`router.py`	Registers `/invoke`, `/stream`, `/schema`, `/health`
`middlewares.py`	Auth, Policy, Telemetry
`adapter.py`	Executes LangGraph / LangChain graphs
`schemas.py`	Pydantic I/O models
`policy.py`	Executes pre/in/post checks
`metrics.py`	Prometheus exposition

7.6.2 Lifecycle Hooks

1. **Startup:** Load DockSpec + LangGraph.
2. **On Request:** Auth → Policy → Adapter.
3. **Shutdown:** Flush telemetry buffers.

7.6.3 Local Storage

- Runtime logs → stdout (JSON).
- Temporary cache for prompt history (10 MB limit).

7.7 Policy Engine

7.7.1 Internal Structure

```
class PolicyEngine:
    def pre_invoke(self, request): ...
    def in_invoke(self, tool_name): ...
    def post_invoke(self, output): ...
```

7.7.2 Config Loader

- Loads `policies` block from Dockfile.
- Compiles regex patterns on startup.

7.7.3 Violation Handler

- Creates structured event:

  ```
  {"type": "violation", "stage": "post", "match": "SSN", "agent":
  "invoice_copilot"}
  ```

- Forwarded to Telemetry Agent.

7.8 Telemetry Agent

7.8.1 Responsibilities

- Capture per-request metrics.
- Emit LangFuse traces and Prometheus metrics.
- Handle fallback to local storage on failure.

7.8.2 Data Contract

Field	Type	Description
run_id	string	Unique trace identifier
agent	string	Agent name
version	string	Active version
latency_ms	int	Total runtime
tokens	dict	`{"in": …, "out": …}`
cost	float	USD estimate

7.9 Dashboard Gateway

7.9.1 GraphQL Schema

Query	Returns
`agents`	List of registered agents
`agent(name)`	Metadata + current metrics
`deployments(agent)`	Deployment history
`logs(agent, filter)`	Paginated log entries

Mutation	Action
`deployAgent(dockfile)`	Triggers Controller deployment
`rollbackAgent(agent, version)`	Initiates rollback
`createApiKey(role)`	Generates API key

7.9.2 Event Channels

Topic	Payload
`build_event`	Progress updates from Builder
`health_event`	Runtime status changes
`metric_event`	Telemetry snapshots

7.10 Frontend UI

7.10.1 Folder Structure

```
/src
 ├─ components/
 ├─ pages/
 ├─ hooks/
 ├─ services/
 ├─ store/
 └─ utils/
```

7.10.2 Data Flow

- GraphQL → React Query → Zustand Store.
- WebSocket events update live metrics.

7.10.3 UI Libraries

- shadcn/ui – cards, tables, tabs.
- Recharts – line & bar charts.
- React Ace – Dockfile YAML editor.
- React JsonView – schema inspector.

7.11 Database Schema (Consolidated)

Table	Purpose	Relations
`agents`	Registered agent metadata	1-N → agent_versions
`agent_versions`	Versioned deployments	N-1 → agents
`deployments`	Active runtime records	N-1 → agent_versions
`api_keys`	Auth tokens	N-1 → agents

Table	Purpose	Relations
`audit_log`	Admin actions log	N-1 → users
`policies`	Parsed policy configs	1-1 → agents
`metrics_cache`	Last reported telemetry	N-1 → agents

All tables versioned with `created_at`, `updated_at`, and `org_id` fields for multi-tenant scalability.

7.12 Queue Layer

Queue	Producer	Consumer	Payload
`build_queue`	Controller Service	Builder Worker	Dockfile + agent metadata
`metrics_queue`	Runtime Service	Telemetry Agent	Metric snapshots
`alerts_queue`	Prometheus / Controller	Dashboard Gateway	Alert notifications

---

7.13 Error & Retry Handling

Component	Strategy
Builder	Retry ×3 → DLQ (`failed_builds`)
Telemetry	Buffer to local file → sync later
Controller	Transaction rollback on DB failure
Runtime	Circuit breaker for model timeouts

7.14 Testing & Validation

Level	Framework	Scope
Unit Tests	Pytest	Each module functionality
Integration Tests	Docker Compose test suite	Controller↔Builder↔Runtime
E2E Tests	Playwright	Dashboard flows

Level	Framework	Scope
Load Tests	Locust	Runtime / API scalability

Target coverage ≥ 85 %.

7.15 Acceptance Criteria

Requirement	Success Definition
Each module deployable independently	Dockerized and CI/CD tested
Interfaces stable	Documented OpenAPI / GraphQL schemas
Metrics propagate end-to-end	Visible in Dashboard & LangFuse
Policy violations handled	No crash, logged + alerted
Full test suite passing	All green in CI pipeline

✅ Completion Definition

The System Components section is complete when:

- Every module has explicit class, data, and API designs.
- Engineers can build or mock any service from this spec alone.
- Cross-service dependencies and message contracts are unambiguous.

---

8. Dockfile Specification Schema (Technical Reference)

This section is the **single source of truth** for Dockfile v1. It includes the formal **JSON Schema**, validation rules, error codes, and reference implementations (Pydantic) to keep parsers, CLIs, and runtimes consistent.

8.1 Versioning & Compatibility

- **Field:** `version: "1.0"` (required)
- **Policy: SemVer** for the spec.

- `1.x` → backward compatible additions only (new optional fields).
- `2.0` → breaking changes (renames, structural moves).

- **Parser rule:** reject files with `version` not recognized by the parser, or < minimal supported version.

8.2 JSON Schema (v1.0)

> Authoritative schema for `Dockfile.yaml` after env interpolation and `!include` expansion.

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "$id": "https://agentdock.io/schema/dockfile-1.0.json",
  "title": "AgentDock Dockfile v1.0",
  "type": "object",
  "required": ["version", "agent", "model", "io_schema", "auth", "expose"],
  "properties": {
    "version": {
      "type": "string",
      "const": "1.0",
      "description": "Dockfile schema version"
    },

    "agent": {
      "type": "object",
      "required": ["name", "entrypoint", "framework"],
      "properties": {
        "name": {
          "type": "string",
          "pattern": "^[a-z][a-z0-9-_]{2,63}$",
          "description": "Kebab/slug name. Lowercase, 3-64 chars."
        },
        "description": { "type": "string" },
        "entrypoint": {
          "type": "string",
          "pattern": "^[a-zA-Z_][\\w\\.]*:[a-zA-Z_][\\w]*$",
          "description": "<module.path>:<factory_or_callable>"
        },
        "framework": {
          "type": "string",
          "enum": ["langgraph", "langchain"],
          "description": "Adapter selection. v1 supports langgraph|langchain."
        }
      },
      "additionalProperties": false
    },
```

```
"model": {
  "type": "object",
  "required": ["provider", "name"],
  "properties": {
    "provider": {
      "type": "string",
      "enum": ["openai", "azure", "anthropic", "google", "ollama", "custom"]
    },
    "name": { "type": "string" },
    "temperature": { "type": "number", "minimum": 0, "maximum": 2 },
    "max_tokens": { "type": "integer", "minimum": 1 },
    "endpoint": { "type": "string" },
    "extra": { "type": "object", "additionalProperties": true }
  },
  "additionalProperties": false
},

"io_schema": {
  "type": "object",
  "oneOf": [
    {
      "required": ["input", "output"],
      "properties": {
        "input": { "$ref": "#/$defs/jsonSchemaObject" },
        "output": { "$ref": "#/$defs/jsonSchemaObject" }
      },
      "additionalProperties": false,
      "description": "Inline JSON-Schema-style definitions"
    },
    {
      "required": ["input_class", "output_class"],
      "properties": {
        "input_class": { "$ref": "#/$defs/pythonDottedPath" },
        "output_class": { "$ref": "#/$defs/pythonDottedPath" }
      },
      "additionalProperties": false,
      "description": "Pydantic class import paths"
    }
  ]
},

"arguments": {
  "type": "object",
  "additionalProperties": {
    "oneOf": [
      { "type": "string" }, { "type": "number" },
      { "type": "integer" }, { "type": "boolean" },
      { "type": "array" }, { "type": "object" }, { "type": "null" }
    ]
```

```
      },
      "description": "Runtime knobs merged into adapter config"
    },

    "policies": {
      "type": "object",
      "properties": {
        "tools": {
          "type": "object",
          "properties": {
            "allowed": {
              "type": "array",
              "items": { "type": "string", "pattern": "^[a-zA-Z0-9_\\-\\.]
{1,64}$" }
            },
            "deny_by_default": { "type": "boolean", "default": true }
          },
          "additionalProperties": false
        },
        "safety": {
          "type": "object",
          "properties": {
            "redact_patterns": {
              "type": "array",
              "items": { "type": "string", "minLength": 1 }
            },
            "max_output_chars": { "type": "integer", "minimum": 1 },
            "block_prompt_injection": { "type": "boolean", "default": true },
            "halt_on_violation": { "type": "boolean", "default": false }
          },
          "additionalProperties": false
        }
      },
      "additionalProperties": false
    },

    "auth": {
      "type": "object",
      "required": ["mode"],
      "properties": {
        "mode": { "type": "string", "enum": ["jwt", "api_key", "oauth2"] },
        "jwt": {
          "type": "object",
          "properties": {
            "secret_key": { "type": "string", "minLength": 16 },
            "expiry_minutes": { "type": "integer", "minimum": 5 }
          },
          "required": [],
          "additionalProperties": false
        },
```

```
        "api_keys": {
          "type": "object",
          "properties": {
            "enabled": { "type": "boolean", "default": false },
            "rotation_days": { "type": "integer", "minimum": 1 }
          },
          "additionalProperties": false
        },
        "roles": {
          "type": "array",
          "items": {
            "type": "object",
            "required": ["name", "permissions"],
            "properties": {
              "name": { "type": "string", "pattern": "^[a-z][a-z0-9-_]{2,32}$" },
              "permissions": {
                "type": "array",
                "items": { "type": "string", "pattern": "^[a-z_]{2,32}$" }
              }
            },
            "additionalProperties": false
          }
        },
        "rate_limits": {
          "type": "object",
          "additionalProperties": {
            "type": "string",
            "pattern": "^[1-9][0-9]*\\s*/\\s*(s|m|h|d)$"
          },
          "description": "e.g., developer: '100/m', viewer: '50/m'"
        }
      },
      "allOf": [
        {
          "if": { "properties": { "mode": { "const": "jwt" } } },
          "then": { "required": ["jwt"] }
        },
        {
          "if": { "properties": { "mode": { "const": "api_key" } } },
          "then": { "properties": { "api_keys": { "properties": { "enabled": {
"const": true } } } } }
        }
      ],
      "additionalProperties": false
    },

    "observability": {
      "type": "object",
      "properties": {
        "langfuse": {
```

```
          "type": "object",
          "properties": {
            "project": { "type": "string" },
            "public_key": { "type": "string" },
            "secret_key": { "type": "string" }
          },
          "required": [],
          "additionalProperties": false
        },
        "tracing": { "type": "boolean", "default": true },
        "log_level": { "type": "string", "enum": ["debug", "info", "warn",
"error"], "default": "info" },
        "metrics": {
          "type": "object",
          "properties": {
            "latency": { "type": "boolean", "default": true },
            "tokens": { "type": "boolean", "default": true },
            "cost": { "type": "boolean", "default": true }
          },
          "additionalProperties": false
        }
      },
      "additionalProperties": false
    },

    "expose": {
      "type": "object",
      "properties": {
        "rest": { "type": "boolean", "default": true },
        "streaming": { "type": "string", "enum": ["sse", "websocket", "none"],
"default": "sse" },
        "port": { "type": "integer", "minimum": 1, "maximum": 65535, "default":
8080 },
        "host": { "type": "string", "default": "0.0.0.0" },
        "cors": {
          "type": "object",
          "properties": {
            "origins": { "type": "array", "items": { "type": "string" } },
            "methods": { "type": "array", "items": { "type": "string" } }
          },
          "additionalProperties": false
        }
      },
      "allOf": [
        {
          "if": {
            "properties": { "rest": { "const": false }, "streaming": { "const":
"none" } },
            "required": ["rest", "streaming"]
          },
```

```
          "then": { "errorMessage": "At least one of rest or streaming must be
enabled." }
        }
      ],
      "additionalProperties": false
    },

    "metadata": {
      "type": "object",
      "properties": {
        "maintainer": { "type": "string" },
        "version": { "type": "string" },
        "tags": { "type": "array", "items": { "type": "string" } }
      },
      "additionalProperties": true
    }
  },

  "$defs": {
    "jsonSchemaObject": {
      "type": "object",
      "required": ["type", "properties"],
      "properties": {
        "type": { "const": "object" },
        "required": {
          "type": "array",
          "items": { "type": "string" }
        },
        "properties": {
          "type": "object",
          "additionalProperties": {
            "type": "object",
            "properties": {
              "type": {
                "type": "string",
                "enum": ["string", "number", "integer", "boolean", "array",
"object", "null"]
              },
              "description": { "type": "string" },
              "items": { "type": "object" },
              "enum": { "type": "array" }
            },
            "required": ["type"],
            "additionalProperties": true
          }
        }
      },
      "additionalProperties": false
    },
    "pythonDottedPath": {
```

```
        "type": "string",
        "pattern": "^[a-zA-Z_][\\w\\.]*:[a-zA-Z_][\\w]*$",
        "description": "module.sub.module:ClassOrFunc"
      }
    },

    "additionalProperties": false
  }
```

> **Note:** We intentionally require either **inline JSON-Schema** or **Python class paths** for `io_schema` via a `oneOf`. This keeps validation strict and unambiguous.

8.3 Environment Variable Interpolation

- Syntax: `"${ENV_NAME}"` inside any **string** field.
- Resolution order: CLI `--env` → process env → `.env` file.
- **Rule:** Unresolved variables cause `4001 EnvInterpolationError` unless `allow_unresolved=true` in CLI (for dry-run previews).

8.4 Includes & Composition

- YAML includes via `!include path/to/file.yaml` are expanded **before** schema validation.
- Circular includes prohibited → `4009 IncludeCycleError`.
- Merged content must still satisfy the schema post-expansion.

8.5 Reserved Words & Naming Conventions

- `agent.name` must be unique per org and **DNS-safe** (`^[a-z][a-z0-9-_]{2,63}$`).
- Reserved permission names: `deploy`, `rollback`, `invoke`, `view_metrics`, `key_manage`, `read_logs`, `read_docs`.
- Tool names in `policies.tools.allowed` must be alnum/underscore/dot/hyphen, ≤ 64 chars.

8.6 Validation Pipeline (CLI/Controller)

1. **Parse:** YAML → Dict; expand `!include`; interpolate env.

2. **Schema Validate:** JSON Schema (this doc).
3. **Semantic Validate:**
   - `entrypoint` importable & callable
   - regex compile for `redact_patterns`
   - auth-mode consistency checks
   - expose sanity (`rest` OR `streaming != none`)
4. **Adapter Validate:** supported `framework` .
5. **Freeze:** normalize + lock to internal `DockSpec` (Pydantic).

8.7 Pydantic Reference Models (authoritative types)

> Minimal excerpt; keep in sync with JSON Schema.

```
from pydantic import BaseModel, Field, constr
from typing import List, Dict, Optional, Literal

NameSlug = constr(pattern=r"^[a-z][a-z0-9-_]{2,63}$")

class AgentCfg(BaseModel):
    name: NameSlug
    description: Optional[str] = None
    entrypoint: constr(pattern=r"^[A-Za-z_][\w\.]*:[A-Za-z_][\w]*$")
    framework: Literal["langgraph", "langchain"]

class ModelCfg(BaseModel):
    provider: Literal["openai", "azure", "anthropic", "google", "ollama",
"custom"]
    name: str
    temperature: Optional[float] = Field(ge=0, le=2, default=None)
    max_tokens: Optional[int] = Field(ge=1, default=None)
    endpoint: Optional[str] = None
    extra: Dict[str, object] = {}

class IOSchemaInline(BaseModel):
    type: Literal["object"] = "object"
    required: Optional[List[str]] = []
    properties: Dict[str, Dict[str, object]]

class IOSchemaCfg(BaseModel):
    input: Optional[IOSchemaInline] = None
    output: Optional[IOSchemaInline] = None
    input_class: Optional[str] = None
    output_class: Optional[str] = None
```

```
    @property
    def mode(self):
        return "inline" if self.input and self.output else "class"

class ToolPolicy(BaseModel):
    allowed: List[constr(pattern=r"^[A-Za-z0-9_.-]{1,64}$")] = []
    deny_by_default: bool = True

class SafetyPolicy(BaseModel):
    redact_patterns: List[str] = []
    max_output_chars: Optional[int] = None
    block_prompt_injection: bool = True
    halt_on_violation: bool = False

class PoliciesCfg(BaseModel):
    tools: Optional[ToolPolicy] = ToolPolicy()
    safety: Optional[SafetyPolicy] = SafetyPolicy()

class RolesCfg(BaseModel):
    name: NameSlug
    permissions: List[constr(pattern=r"^[a-z_]{2,32}$")]

class AuthJwtCfg(BaseModel):
    secret_key: Optional[constr(min_length=16)] = None
    expiry_minutes: Optional[int] = Field(ge=5, default=60)

class ApiKeysCfg(BaseModel):
    enabled: bool = False
    rotation_days: Optional[int] = Field(ge=1, default=None)

class AuthCfg(BaseModel):
    mode: Literal["jwt", "api_key", "oauth2"]
    jwt: Optional[AuthJwtCfg] = None
    api_keys: Optional[ApiKeysCfg] = None
    roles: List[RolesCfg] = []
    rate_limits: Dict[str, constr(pattern=r"^[1-9][0-9]*\\s*/\\s*(s|m|h|d)$")] =
{}

class ExposeCors(BaseModel):
    origins: List[str] = []
    methods: List[str] = []

class ExposeCfg(BaseModel):
    rest: bool = True
    streaming: Literal["sse", "websocket", "none"] = "sse"
    port: int = 8080
    host: str = "0.0.0.0"
    cors: Optional[ExposeCors] = None
```

```python
class ObservabilityCfg(BaseModel):
    langfuse: Optional[Dict[str, str]] = None
    tracing: bool = True
    log_level: Literal["debug", "info", "warn", "error"] = "info"
    metrics: Dict[str, bool] = {"latency": True, "tokens": True, "cost": True}

class MetadataCfg(BaseModel):
    maintainer: Optional[str] = None
    version: Optional[str] = None
    tags: List[str] = []

class DockSpec(BaseModel):
    version: Literal["1.0"]
    agent: AgentCfg
    model: ModelCfg
    io_schema: IOSchemaCfg
    arguments: Dict[str, object] = {}
    policies: Optional[PoliciesCfg] = PoliciesCfg()
    auth: AuthCfg
    observability: Optional[ObservabilityCfg] = ObservabilityCfg()
    expose: ExposeCfg
    metadata: Optional[MetadataCfg] = MetadataCfg()
```

8.8 Error Codes (validation & semantics)

Code	Name	When it fires
`4001`	EnvInterpolationError	`${VAR}` unresolved and not allowed
`4002`	InvalidSchemaError	JSON Schema validation failed
`4003`	EntrypointNotFoundError	Import path or callable missing
`4004`	InvalidIOSchemaError	Inline schema malformed / class import fails
`4005`	AuthConfigError	Mode inconsistent (e.g., `jwt` without secret)
`4006`	PolicyValidationError	Regex compile error / bad tool name
`4007`	UnsupportedFrameworkError	`framework` not supported
`4008`	TelemetryConfigError	LangFuse required but keys missing
`4009`	IncludeCycleError	`!include` cycles or depth exceeded
`4010`	ExposureError	`rest=false` and `streaming=none`

All errors must include **path hints** (e.g., `auth.jwt.secret_key`) and a **single-line cause** for CI readability.

8.9 Lint Rules (non-fatal warnings)

- Warn if:
  - `redact_patterns` contains overlapping or catastrophic backtracking regex.
  - `temperature > 1.2` with `block_prompt_injection=false`.
  - Missing `roles` for non-`api_key` mode.
  - `max_output_chars` is unset for public agents.
- CLI flags: `agentdock validate --strict` elevates warnings to errors.

8.10 Minimal / Full Examples

Minimal (API key, inline schema):

```yaml
version: "1.0"
agent:
  name: doc_summarizer
  description: "Summarize documents"
  entrypoint: src.agent:build_graph
  framework: langgraph
model:
  provider: openai
  name: gpt-4o-mini
io_schema:
  input:
    type: object
    properties: { document: { type: string } }
  output:
    type: object
    properties: { summary: { type: string } }
auth:
  mode: api_key
  api_keys: { enabled: true }
expose:
  rest: true
```

Full (JWT, policies, SSE, LangFuse):

```yaml
version: "1.0"
agent:
  name: invoice_copilot
  description: "Extract & summarize invoices"
  entrypoint: app.graph:build_graph
  framework: langgraph
model:
```

```
  provider: azure
  name: gpt-4o
  temperature: 0.2
  max_tokens: 2000
  endpoint: https://api.openai.azure.com
io_schema:
  input_class: app.schemas.InvoiceInput
  output_class: app.schemas.InvoiceOutput
arguments:
  timeout_sec: 20
  enable_streaming: true
policies:
  tools: { allowed: [extract_invoice, summarize_invoice], deny_by_default: true }
  safety:
    redact_patterns: ["\\d{16}"]
    max_output_chars: 5000
    block_prompt_injection: true
auth:
  mode: jwt
  jwt:
    secret_key: "${JWT_SECRET}"
    expiry_minutes: 60
  api_keys: { enabled: true, rotation_days: 30 }
  roles:
    - name: admin
      permissions: [deploy, rollback, invoke, view_metrics, key_manage]
    - name: developer
      permissions: [invoke, view_metrics, read_logs]
observability:
  langfuse:
    project: invoice-copilot
    public_key: "${LANGFUSE_PUBLIC}"
    secret_key: "${LANGFUSE_SECRET}"
  tracing: true
  log_level: info
expose:
  rest: true
  streaming: sse
  port: 8080
metadata:
  maintainer: "paritosh.sharma@company.com"
  tags: [finance, extraction]
```

8.11 Test Vectors (golden cases)

- ✅ **Happy path:** full example above.

- ❌ Missing `agent.entrypoint` → `4002 InvalidSchemaError`.
- ❌ `auth.mode=jwt` without `jwt.secret_key` → `4005 AuthConfigError`.
- ❌ `expose.rest=false` and `expose.streaming=none` → `4010 ExposureError`.
- ❌ `policies.safety.redact_patterns` contains invalid regex → `4006 PolicyValidationError`.
- ❌ `io_schema` provides `input_class` but not `output_class` → `4004 InvalidIOSchemaError`.

8.12 Acceptance Criteria

- JSON Schema passes validation for all **golden examples** and rejects **negative tests** above.
- CLI `agentdock validate` runs steps 8.6 (parse→freeze) and outputs machine-readable errors (`--json`).
- Controller performs identical validation to prevent drift with CLI.
- Reference Pydantic models remain **byte-for-byte consistent** with JSON Schema (contract tests).

---

9. Non-Functional Requirements (NFRs)

This section sets **platform-wide SLOs and constraints** that every AgentDock v1 component must satisfy. Each item includes **measurable targets**, **verification methods**, and **ownership** to keep teams aligned.

9.1 Performance

Area	Target (v1)	Measurement	Notes
Cold start (runtime boot)	≤ **2 s** p95	synthetic probe hitting `/health` after pod start	Base image: `python:3.11-slim`, lazy init of adapters
Invoke latency overhead	≤ **10%** vs raw LangGraph	A/B harness (adapter only vs runtime)	Same prompt/input & model
p95 invoke latency	≤ **1.8 s** (model-dependent)	Locust load test (RPS=5/agent)	Exclude LLM queue time if provider throttles
Deploy time	≤ **90 s** p95	CLI timed from `deploy` to healthy	Includes build + push + start

Area	Target (v1)	Measurement	Notes
Rollback time	≤ **30 s** p95	CLI stopwatch + health	Image pull warm cache assumption
Throughput	≥ **50 RPS**/agent runtime	Locust ramp to sustained load	With streaming off; on = ≥20 RPS

Acceptance: Load test reports attached to release; deltas tracked per commit.

9.2 Reliability & Availability

Area	Target	Measurement	Notes
Runtime availability	**99.5%** monthly	black-box probes	Single zone; HA optional
Controller availability	**99.9%** monthly	internal health checks	Stateless, ≥2 replicas
MTTR	< **60 s**	synthetic failures (kill pod)	Auto restart + health gate
Error budget policy	0.5% monthly	SLO dashboard	Triggers deploy freeze if breached

Acceptance: SLO dashboard visible in the Dashboard “System Status” card.

9.3 Scalability

Dimension	v1 Capability	Path to Scale
Agents per org	**≤ 100**	Controller + Postgres horizontal scale
Concurrent runtimes	**≤ 200 pods**	K8s HPA; per-agent autoscale rules
Deploy concurrency	**≤ 20** parallel builds	Builder workers via Redis queue
Metrics volume	**≤ 5k spans/min**	LangFuse SaaS; local back-pressure buffer

Acceptance: Stress test with 50 agents × 2 replicas passes SLOs.

9.4 Security

Control	Requirement (v1)	Verification
Transport security	TLS 1.3 everywhere; HSTS on public endpoints	SSL scans, config checks
Auth	JWT (HS256) and API keys; rotation + revocation	Unit + integration tests
Secrets	Stored in K8s Secrets/Vault; never in images/logs	CI secret scan; image trivy
Least privilege	Per-endpoint permission checks; tool allowlist	AuthZ tests in CI
PII protection	Redaction pre/post invoke; regex compile on startup	Policy tests + e2e traces
Audit	All admin actions in `audit_log`	DB assertions; UI audit timeline

Acceptance: Security checklist signed; automated scanners pass (Trivy, Gitleaks).

9.5 Observability

Area	Requirement	Verification
Traces	100% of runs create LangFuse trace with `agent`, `version`, `role`	Golden test comparing counts
Metrics	`/metrics` Prometheus endpoint; latency, req/s, error rate	Promtool lint + scrape success
Logs	JSON structured logs; include `trace_id`	Log schema linter; sample replay
Correlation	dashboard deep-link from request → LangFuse trace	Manual e2e click-through
Fallback	Local buffer if LangFuse down; 7-day retention	Fault-injection test

Acceptance: Observability smoke test included in CI; dashboards pre-wired.

9.6 Maintainability & Modularity

Area	Target	Verification
Code coverage	**≥ 85%** unit coverage	CI gate

Area	Target	Verification
Cyclomatic complexity	< 15 per function (hot paths)	Lint job (radon)
Public interfaces	Stable typed contracts (OpenAPI/GraphQL)	Contract tests; semver
Adapter boundary	LangGraphAdapter replaceable behind interface	Mock adapter test suite

Acceptance: All repos pass lint/format/type (ruff, black, mypy).

9.7 Portability & Deployability

Area	Requirement	Verification
Container standard	OCI-compliant images; non-root user	Dockerfile inspection
Runtime	Runs on Docker & K8s (AKS/GKE/EKS)	Matrix deploy workflow
Config	12-factor: env-driven; Dockfile as single source	Dry-run `validate` parity with Controller

Acceptance: “Getting Started” Compose and Helm chart both green.

9.8 Usability (DX & UX)

Area	Target	Verification
CLI ergonomics	Clear progress bars + JSON mode; exit codes deterministic	Snapshot tests
Docs	Autogenerated `/docs` + `/openapi.json`; copy-paste examples	Doc lints; link check
Dashboard	Page TTI ≤ **2 s**; live deploy stream ≤ **500 ms** delay	Web-vitals CI

Acceptance: Dev survey ≥ **8/10** usability in pilot.

9.9 Compliance & Data Handling (foundational)

Area	v1 Stance	Notes
Data residency	Not enforced; can pin LangFuse self-hosted	v2 add residency controls
PII	Best-effort redaction; no persistent user payloads	Only metadata in Postgres
Retention	Logs 14d (configurable); metrics 30d	Admin can purge per agent

Acceptance: Retention knobs exposed in Admin Panel.

9.10 Cost & Efficiency

Area	Target	Verification
Image size	≤ **350 MB** runtime image	CI artifact check
CPU	Idle < **50m**, p95 invoke < **500m**	Prometheus
RAM	Idle < **300 MB**, p95 invoke < **1.2 GB**	Prometheus
Telemetry overhead	≤ **5%** additional latency	A/B benchmark

Acceptance: Cost dashboard shows $/1k calls from tokens × pricing table.

9.11 Internationalization & Accessibility (MVP baseline)

Area	Requirement	Verification
i18n	UI copy externalized; English default	String extractor pass
A11y	WCAG 2.1 AA for core flows; keyboard nav	Lighthouse a11y ≥ 90

Acceptance: Lighthouse reports attached to release candidate.

9.12 Disaster Recovery & Backups

Asset	Policy	RPO/RTO
Postgres	Daily snapshot; PITR if managed	RPO 24h / RTO 30m
Registry	Images immutable + retained 30d	N/A
Config (Dockfiles)	Stored per version in DB + git tag optional	Dual copy

Acceptance: Quarterly DR drill restores v1 environment successfully.

9.13 Non-Goals (NFR scope v1)

- Formal SOC2/ISO audits (design for later).
- Multi-region active-active.
- Per-tenant hard isolation (namespaces only in v1).
- Real-time cost billing (show estimates; no chargeback).

9.14 Validation Plan (How we prove NFRs)

1. **Perf suite** (Locust + k6): latency/throughput/cold-start baselines per commit.
2. **Chaos tests** (pod kill, LangFuse outage, DB failover): MTTR & fallback.
3. **Security scans** (Trivy, Gitleaks, OPA policy check).
4. **Contract tests** for OpenAPI/GraphQL; breaking changes fail CI.
5. **Soak test** 24h with 20 agents × 5 RPS; error budget tracking.

9.15 Acceptance Criteria (Overall NFR sign-off)

- SLO dashboards show green for **Performance, Availability, Error Budget** over a 7-day burn-in.
- All CI quality gates (lint, type, tests, security, contract) pass.
- DR drill + chaos tests meet **RPO/RTO** and **MTTR** targets.
- Observability confirms **100% trace coverage** and valid `/metrics`.
- Usability and image-size/cost targets are met.

---

10. Roadmap Overview

This section maps the **strategic rollout plan** for AgentDock across versions, balancing feature maturity, developer adoption, and enterprise-readiness. Each milestone reflects a cohesive product vision — from *single-agent deployment* to *multi-agent orchestration* and *enterprise AgentOps management.*

10.1 Product Evolution Stages

Stage	Codename	Core Focus	Outcome
v1.0	*Single Dock*	One-click agent deployment via Dockfile	Developer productivity + baseline security
v1.1	*Fleet Control*	Teams, org-level controls, and improved monitoring	Enterprise readiness
v2.0	*AgentOps*	Multi-agent workflows, DAGs, and dependency resolution	Production-scale orchestration
v3.0	*Agent Mesh*	Federated runtime clusters + auto-scaling	Autonomous agent networks (SaaS-ready)

10.2 v1.0 – Core MVP (AgentDock Foundation)

Timeline: 8–10 weeks
Goal: Ship the minimal viable, secure, and observable deployment platform for single agents.

Scope

✅ Dockfile v1 (YAML spec + schema validation)
✅ Controller, Builder, Auth, and Runtime services
✅ CLI (`agentdock deploy` , `agentdock validate` , `agentdock logs`)
✅ Observability via LangFuse + Prometheus
✅ React dashboard (deploy, monitor, rollback)
✅ JWT/API key authentication
✅ SSE streaming for logs and deploy events

Non-Goals

- Multi-tenant orgs
- Advanced billing
- Multi-agent DAGs

Deliverable:
A developer can deploy a single LangGraph/LangChain agent as a secure FastAPI service in under 90 seconds.

10.3 v1.1 – Fleet Control (Team and Observability Upgrade)

Timeline: +8 weeks post v1
Goal: Enhance multi-user collaboration, auditability, and monitoring depth.

New Features

🟢 **Organization & Teams**

- Multi-tenant orgs with RBAC and roles
- Org-level API keys and rate limits

🟢 **Enhanced Monitoring**

- Per-agent SLA and latency dashboards
- Token and cost breakdown (per org)
- Alerting hooks (Slack, Webhook, Email)

🟢 **Lifecycle Enhancements**

- Soft-delete for agents and versions
- Deployment freeze/unfreeze toggle
- Dockfile diff view (change audit)

Tech Enhancements

- Postgres schema separation per org
- LangFuse project mapping per org
- SSO (OAuth2) for enterprise users

Outcome:
Team-ready platform for small and mid-sized enterprises to manage agent fleets securely and collaboratively.

10.4 v2.0 – AgentOps (Multi-Agent Orchestration Platform)

Timeline: 12–14 weeks after v1.1
Goal: Enable *complex agentic workflows* — multiple agents interacting with shared context and dependencies.

New Capabilities

⚙️ Workflow Orchestration

- YAML-defined multi-agent DAGs
- Directed edges (dependencies) with data passing
- Runtime orchestrator managing stateful interactions

🧩 Shared Context Management

- Shared memory per workflow (Redis / vector store)
- Scoped state (ephemeral vs persistent)
- Cross-agent data contracts (validated schemas)

📈 Observability Extensions

- Cross-agent trace stitching in LangFuse
- Workflow-level latency, cost, and SLA metrics

🔐 Security

- Policy enforcement at DAG node level
- Scoped secrets & data isolation

Example:

```
workflow:
  name: legal_chain
  agents:
    - name: contract_parser
      output: parsed_terms
    - name: risk_analyzer
      input: parsed_terms
    - name: summarizer
      input: risk_summary
  flow: [contract_parser -> risk_analyzer -> summarizer]
```

Outcome:
AgentDock transitions from *deployment platform → agent orchestration engine* for enterprise automation.

10.5 v3.0 – Agent Mesh (Federated & SaaS Ready)

Timeline: 16–18 weeks after v2
Goal: Build a *federated, distributed runtime mesh* enabling cross-cluster agent networking, observability, and governance — SaaS multi-tenant release.

New Pillars

🌐 Federated Mesh Runtime

- Service mesh connecting agent runtimes (Envoy / Istio)
- Global registry for discoverable agents
- Runtime peer auth + request routing

🔄 Dynamic Scaling

- Autoscaling per agent workload
- Cold-start avoidance via warm pods
- Global load balancer for multi-region clusters

📡 Enterprise Integrations

- OpenTelemetry integration for observability unification
- Billing and cost center reporting
- Compliance: SOC2-ready audit trails

🚀 Self-Service SaaS Portal

- Tenant onboarding, billing, credit usage
- API tokens and private registries
- Usage-based billing backend

Outcome:
AgentDock becomes a *cloud-native AgentOps SaaS* — enabling enterprises and developers to deploy, orchestrate, and monitor agents at scale.

10.6 Cross-Version Architectural Principles

Principle	Description
Spec-first	Dockfile drives deployment deterministically (infra as spec).
Composable microservices	All modules (Controller, Builder, Auth, Runtime) stay independently deployable.

Principle	Description
Pluggable Observability	LangFuse now, OpenTelemetry later.
Security by default	JWT, RBAC, secrets vaulting baseline for all releases.
Future-ready API	All internal APIs contract-tested for backward compatibility.
Multi-cloud support	AKS/GKE/EKS agnostic; registry & secrets modular.

10.7 Dependencies & Risks

Category	Dependency	Risk	Mitigation
Infra	Docker build & registry access	Build delays / size limits	Layer caching, slim base
Observability	LangFuse API stability	SDK rate limits	Buffer & retry mechanism
Security	Vault / K8s secret integration	Misconfig leak	Config validator at build-time
LangGraph SDK	Upstream API changes	Adapter breakage	Interface abstraction layer
Frontend	GraphQL latency under load	Slow UI updates	BFF caching layer
Adoption	Dev onboarding friction	Drop in trial conversion	Templates + CLI wizard

---

10.8 Long-Term Vision

> 🧭 *AgentDock → the "Vercel for Agents"*

A platform where:

- Developers push any agent repo with a Dockfile,
- AgentDock builds, secures, and hosts it automatically,
- Enterprises monitor, govern, and scale agent fleets safely,
- Agents collaborate seamlessly across org boundaries through the **Agent Mesh**.

---

11. Future Enhancements & v2+ Concepts

11.1 Objective

To outline **forward-looking capabilities** beyond v1 that expand AgentDock's scope from individual agent deployment to **autonomous multi-agent orchestration, vector-state persistence, cost intelligence, and adaptive scaling.**

These are **design-level proposals**, not commitments — meant to inform architectural decisions in v1 so we can scale without redesign.

11.2 DAG Workflows & Agent Pipelines

Problem

Today, every agent is deployed in isolation. Enterprise automation often requires **sequential / conditional orchestration** across multiple agents.

Vision

Introduce a **Workflow Engine** capable of executing YAML-defined DAGs (Directed Acyclic Graphs) describing how agents interact and pass outputs.

```yaml
workflow:
  name: contract_review
  nodes:
    - id: parser
      agent: doc_parser
    - id: analyzer
      agent: risk_assessor
      depends_on: [parser]
    - id: summarizer
      agent: summary_bot
      depends_on: [analyzer]
  edges:
    - parser.output -> analyzer.input
    - analyzer.output -> summarizer.input
```

Core Concepts

Element	Description
Node	An existing deployed agent (Dockfile runtime).
Edge	Data flow with I/O schema validation.

Element	Description
Execution Context	Shared state (memory + temp store).
Supervisor	Manages concurrency, retries, rollback.

Implementation Preview

- `WorkflowController` service schedules and orchestrates nodes.
- Uses **Redis Streams** or **Temporal.io** for event-driven coordination.
- Provides DAG visualizer in dashboard with step-level telemetry (LangFuse trace stitching).

11.3 Vector State & Shared Memory

Goal

Enable agents within a workflow (or across invocations) to **retain context and share embeddings** securely.

Design

- **Vector Adapter Interface** (pluggable):

```
class VectorStoreAdapter(Protocol):
    def upsert(self, namespace, embeddings, metadata): ...
    def query(self, namespace, query_vec, top_k): ...
```

- Supported backends: **Postgres pgvector**, **Chroma**, **Milvus**, **Azure AI Search**.
- **Namespace Isolation:** `org_id / agent_id / version` for tenancy safety.
- **Persistence Modes:**
  - *Ephemeral:* in-memory for single run.
  - *Persistent:* cross-workflow retention with TTL policies.

Outcome

Multi-agent workflows can operate over **shared semantic memory**, enabling context-aware orchestration and adaptive behaviors.

11.4 Cost Analytics & Optimization Engine

Objective

Expose **real-time token and cost intelligence** for enterprises.

Features

- Token breakdown by agent, version, role, workflow.
- USD estimation via provider pricing tables.
- Anomaly alerts (cost spike > threshold).
- Forecasting using Prophet or ARIMA.
- Cost-per-agent dashboards and “top spenders.”

Architecture

- `CostAnalyzer` microservice consumes LangFuse events.
- Normalizes into a `cost_ledger` table in Postgres.
- Feeds data to Grafana panels or Dashboard charts.

11.5 Adaptive Scaling & Predictive Autoscaling

Problem

Static HPA rules can’t anticipate LLM latency variance or token bursts.

Solution

Introduce **Adaptive Autoscaler** that learns usage patterns.

Input Signal	Data Source	Action
RPS Spike	Prometheus metrics	Scale replicas ↑
Latency Drift	LangFuse trace stats	Pre-warm pods
Cost Rate	CostAnalyzer	Cool down non-critical agents
Workflow Dependency	DAG Supervisor	Co-scale linked nodes

Implementation: ML-based forecast (Prophet/LSTM) inside the Autoscaler controller; uses K8s API for replica set adjustments.

11.6 Policy Graph & Governance Layer

Concept

Move beyond static Dockfile policies to a **Policy Graph** — an explicit model of data flows, tools, and roles.

- Policies become edges between subjects (users), resources (agents/tools), and actions (invoke/deploy).
- Evaluated via OPA (Open Policy Agent) + Rego rules.
- Enables audit queries like:

> "Which agents can access PII tools in org X?"

11.7 Agent Testing & Shadow Deployment

Purpose

Reduce risk before production deployment.

Features

- **Shadow Mode:** mirror traffic to new version; compare outputs.
- **A/B Evaluation:** split percentage traffic across versions.
- **Golden Tests:** deterministic input/output pairs validated in CI.
- **Regression Dashboard:** metric delta on latency / accuracy.

Implementation

- Traffic router in Runtime Gateway.
- Evaluation engine integrated with LangFuse analytics.

11.8 Agent Marketplace (v3 vision)

A public/private **registry of verified Dockfiles and prebuilt agents.**

- Developers publish Dockfile + metadata.
- Consumers deploy via `agentdock pull <agent>`.
- Includes rating, version badge, and security signature.
- Enterprise mode: internal curated registry behind SSO.

11.9 OpenTelemetry & Unified Observability

Goal

Standardize all telemetry (LangFuse + Prometheus + Logs) into **OpenTelemetry traces**.

- One trace per agent invocation → propagated across workflows.
- Compatibility with Grafana Tempo / DataDog / Elastic APM.
- Enables enterprise visibility across agents, models, and tools.

11.10 Compliance & Enterprise Features (≥ v3)

Feature	Description
SOC2/ISO 27001 alignment	Log integrity, incident reporting pipelines
SAML SSO / OIDC	Enterprise SSO integration
Audit Trail Immutability	WORM-store for audit logs
Custom SLAs	Per-org uptime contracts and alerts
Data Residency Zones	Deploy agents in region-specific clusters

---

11.11 Extensibility SDK

Expose a Python SDK and CLI extensions API.

- `agentdock plugin create <name>` → scaffolds plugin repo.
- Hooks for custom validators, runtimes, storage backends.
- Enables open-source ecosystem growth.

11.12 Long-Term Strategic Endgame

> "AgentDock = Vercel + Airflow + Datadog for LLM Agents."

Layer	Purpose
Spec (L1)	Dockfile as declarative contract
Deploy (L2)	Build + runtime automation
Orchestrate (L3)	DAG and multi-agent management
Observe (L4)	Unified metrics + trace analytics
Govern (L5)	Policies, compliance, cost governance

Layer	Purpose
Monetize (L6)	Marketplace and billing layer

11.13 Dependencies for Future Versions

Module	Dependency Stack	Future Compatibility
Workflow Engine	Temporal.io / Argo / Dagster	v2
Vector Adapter	pgvector / Chroma / Milvus	v2
Cost Analyzer	LangFuse API + OpenAI pricing feed	v2
Autoscaler	K8s HPA + Custom Metrics API	v2 → v3
Policy Graph	OPA + Rego	v3
Marketplace	Registry API + Artifact signing	v3
OpenTelemetry	OTLP Protocol	v3

11.14 Acceptance Criteria for Future Readiness

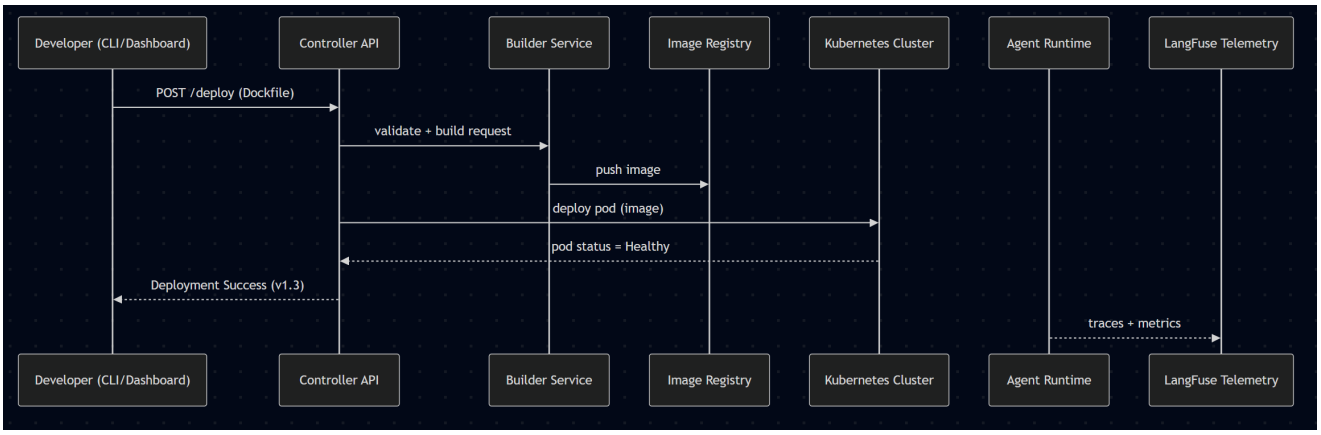
- v1 architecture supports plugin modules without schema break.
- All critical services (Controller, Builder, Runtime) emit OpenTelemetry events.
- Dockfile v1 fields mapped forward to v2 (no migration pain).
- Vector store adapter interface already implemented behind feature flag.
- ML autoscaler prototypes tested offline with LangFuse data.

---

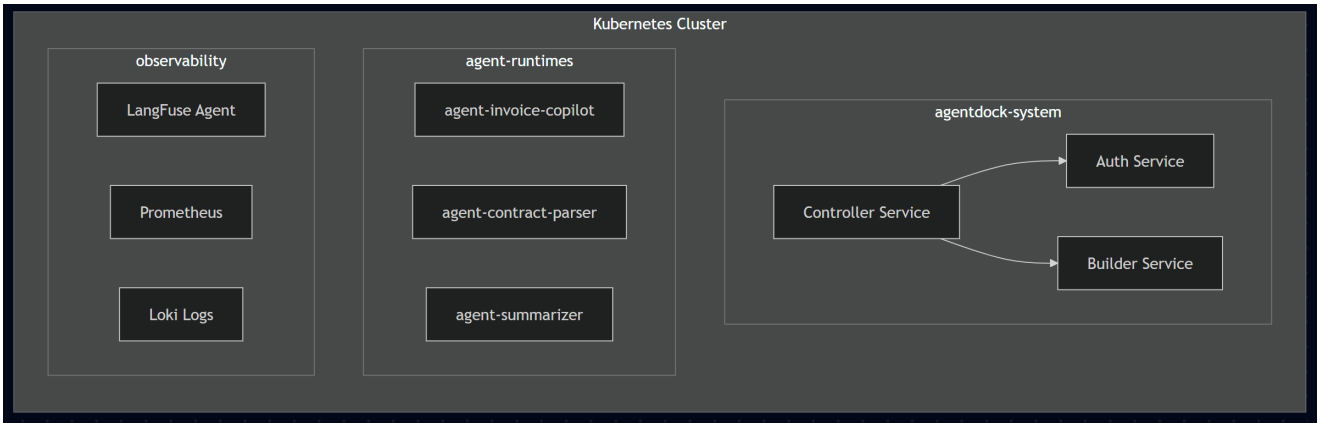
12. Appendices & Artifacts

12.1 Deployment Flow Diagram

> End-to-end picture of how a Dockfile becomes a live runtime.



12.2 Runtime Topology (Infra View)



Notes

- `agentdock-system` = control-plane services
- `agent-runtimes` = isolated namespaces per org or agent
- `observability` = optional external stack, connected via OTLP

12.3 Observability Stack

Source	Collector	Store / UI
FastAPI / Runtime logs	Loki + Promtail	Grafana Loki
LangGraph traces	LangFuse SDK	LangFuse UI
Metrics (/metrics)	Prometheus	Grafana Dashboards
OpenTelemetry export	OTLP Collector	Tempo / DataDog

> Unified correlation key → `trace_id` propagated across all three layers.

12.4 API Endpoint Catalog

Controller API

Method	Path	Description
POST	`/controller/deploy`	Trigger build & deployment
POST	`/controller/rollback`	Revert to previous version
GET	`/controller/status/{agent}`	Health & version status
WS	`/controller/events`	Stream build/deploy logs

Builder API

Method	Path	Description
POST	`/build/start`	Queue Dockfile build job
GET	`/build/status/{id}`	Check build progress

Auth API

Method	Path	Description
POST	`/auth/token`	Issue JWT
POST	`/auth/verify`	Validate JWT / API key
POST	`/auth/key/create`	Generate API key
DELETE	`/auth/key/revoke`	Revoke API key

Runtime API

Method	Path	Description
POST	`/invoke`	Execute agent on input schema
GET	`/health`	Liveness check
GET	`/schema`	Return input/output schema
GET	`/metrics`	Prometheus endpoint
SSE	`/stream`	Streaming response channel

Dashboard GraphQL

Query / Mutation	Purpose
`agents`	List registered agents
`agent(name)`	Get metadata + metrics
`deployAgent(dockfile)`	Deploy agent
`rollbackAgent`	Rollback version
`createApiKey(role)`	Generate new key

12.5 Event Schemas

BuildEvent

```
{
  "event": "build_progress",
  "build_id": "bld_93fe2",
  "status": "RUNNING",
  "percent": 42,
  "message": "Installing dependencies..."
}
```

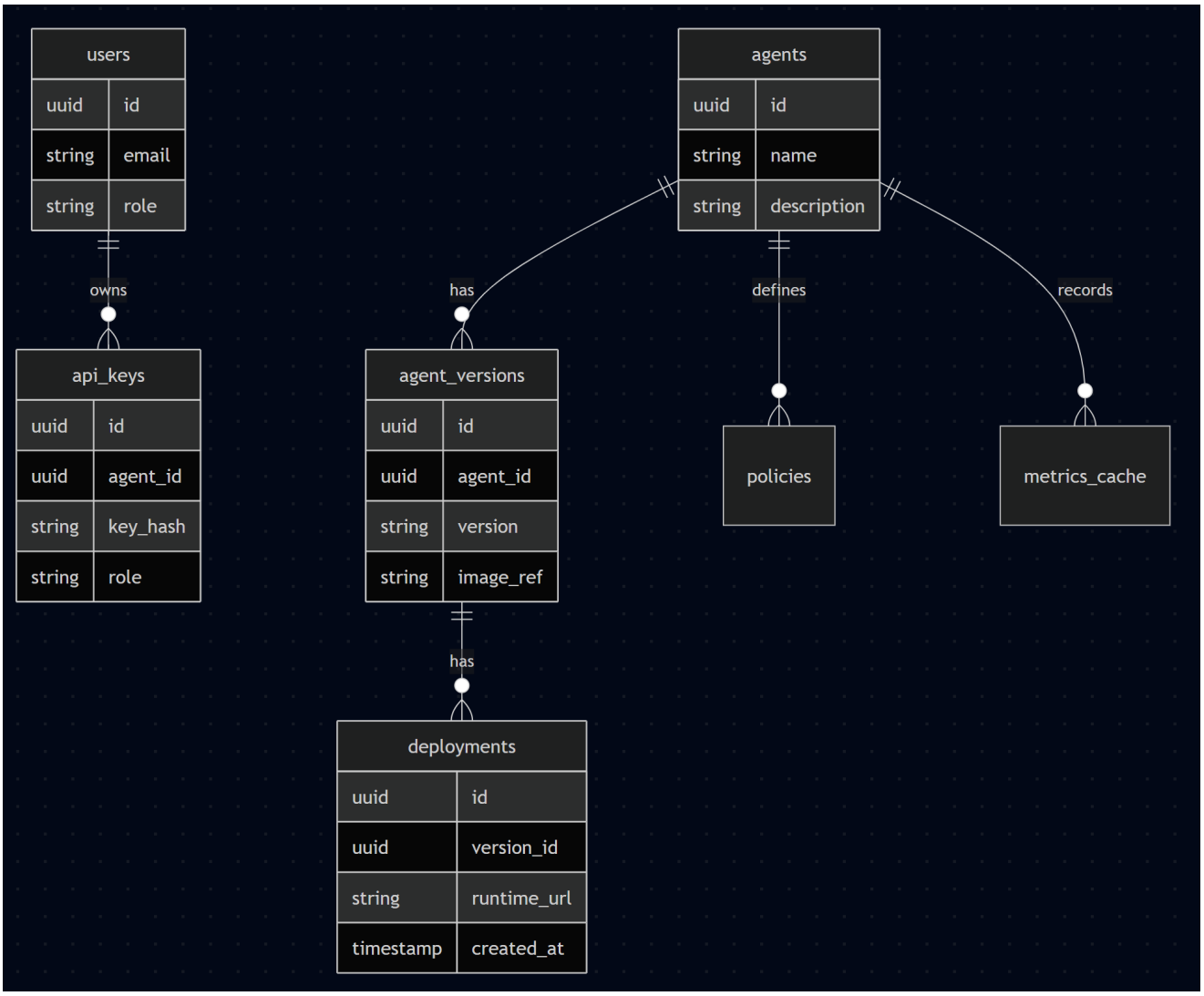
DeployEvent

```
{
  "event": "deploy_status",
  "agent": "invoice_copilot",
  "version": "1.0.3",
  "status": "Healthy",
  "runtime_url": "https://agents.acme.ai/invoice_copilot"
}
```

TelemetryEvent

```
{
  "trace_id": "ad3c1f12",
  "agent": "invoice_copilot",
  "latency_ms": 1634,
  "tokens": { "in": 1200, "out": 300 },
  "cost_usd": 0.0043
}
```

12.6 Database Schema Diagram



12.7 Error Codes Reference (Combined)

Code	Subsystem	Meaning
`1001`	Controller	Invalid Dockfile syntax
`1002`	Builder	Docker build failure
`1003`	Auth	Token expired / revoked
`1004`	Runtime	Schema validation failed
`1005`	Runtime	Policy violation blocked output
`1006`	Dashboard	Unknown GraphQL field
`4001` – `4010`	Dockfile Validator	Validation errors (see Sec 8)

12.8 Glossary

Term	Meaning
Dockfile	YAML spec that defines agent deployment contract
Agent	Runnable LangGraph/LangChain graph exposed as API
Controller Service	Central orchestrator managing deploy/rollback
Builder Service	Compiles Dockfile → runtime image
Runtime	Auto-generated FastAPI server executing agent
LangFuse	Telemetry tool for traces, tokens, costs
Workflow	Multi-agent directed execution graph (v2+)
Policy Engine	Layer enforcing tool and redaction rules
AgentOps	Operational discipline for managing agents in production
Agent Mesh	Federated network of agents (v3+)
Vector State	Persistent semantic memory for agents (v2+)

12.9 Naming & Versioning Standards

Entity	Convention	Example
Agent Name	kebab-case [3-64 chars]	`invoice-copilot`
Docker Image	`org/agent:version`	`acme/invoice-copilot:v1.2.0`
Runtime Container	`agentdock-<agent>-<hash>`	`agentdock-invoice-a13f`
Env Vars	UPPER_SNAKE	`JWT_SECRET_KEY`
Version SemVer	MAJOR.MINOR.PATCH	`1.0.3`
Branch Naming	`feature/` , `fix/` , `chore/`	`feature/vector-adapter`

12.10 CI/CD Workflow (Simplified)

```yaml
name: Build & Deploy
on:
  push:
    branches: [main]
jobs:
  build:
    runs-on: ubuntu-latest
```

```
    steps:
      - uses: actions/checkout@v4
      - name: Validate Dockfile
        run: agentdock validate ./Dockfile.yaml
      - name: Build Docker Image
        run: docker build -t $IMAGE_TAG .
      - name: Push to Registry
        run: docker push $IMAGE_TAG
  deploy:
    needs: build
    steps:
      - name: Deploy to K8s
        run: agentdock deploy ./Dockfile.yaml --cluster=prod
```

12.11 Release Artifacts

Artifact	Format	Purpose
Dockfile Schema	`dockfile-1.0.json`	Validation contract
API Spec	`openapi.yaml`	REST contract for Controller/Runtime
GraphQL Schema	`dashboard.graphql`	Frontend integration
Helm Chart	`agentdock-chart/`	K8s deployment
CLI Binary	`agentdock-cli-v1.x.x`	Developer tool
Sample Repo	`examples/invoice_copilot/`	Reference agent

12.12 Future Appendices (v2+)

Planned additions once multi-agent and marketplace features land:

- **Workflow Spec Schema (JSON)**
- **Vector Adapter Interface Docs**
- **Cost Analyzer Data Model**
- **Autoscaler Algorithm Paper**
- **Policy Graph DSL**

✅ Completion Definition

This PRD is **implementation-ready** when:

- All 12 sections (Overview → Appendices) are approved.
- Reference diagrams, APIs, and schemas are version-locked.
- Engineering can build v1 end-to-end using only this document.
- Future versions (v2 AgentOps, v3 Mesh) remain architecturally compatible.
