QAI_Agents_Framework-Notes

Background:

The quantum and AI paradigms has made rapid progress, especially the advent of the Transformer model and breakthrough in quantum algorithms have offered immense opportunities to explore and leverage its benefits in modernizing the existing industries and economic conditions using highly productive technologies and experts in these areas.

At our startup we are advancing in leveraging both of these and have been developing various concepts in areas of QAI products, frameworks, tools and demos.

In continuation to this, we have plans to design the QAI Agents framework and have sourced various research works in this area. We plan to adopt several concepts, themes and content from these.

We also have our home grown frameworks like Org framework, 16 Ops frameworks, and a business transformation framework that is highly customizable and configurable.

We need a framework that uses both quantum and classical features, functions and operating environments. What we finally need is a deployable unit that can be executed in various popular operating systems like Win, Linux, MacOS etc and use various popular thin or thick clients like browsers, mobile apps, Desktop apps etc. Initial focus will be Win OS and Browser client.

We plan to make best use of our existing home grown products in developing this executable that can be something like a Container that can be deployed easily and maintained by Kubernetes cluster.

We need various environment files, manifests, config files, executable units, that can progress from product idea to product design, prototype, implementation, integration, testing and deployment. We need to honor various lifecycle models of QAI and related guardrails and principles .

We plan to use Classical Agentic AI and Quantum Agent frameworks as proposed in the references, code preferred is Python.

Thin clients need to be designed may be using Generative AI like browser pages using Javascript like React etc in order to be executed at client location

Server will be the Bhadale QAI Hub that hosts most of the Enterprise Core systems, libraries, master data and content that is the innermost layer of the onion structure.

This framework will make various types of calls like RPC, classical queues, ID verifications, backend calls to quantum hardware, helper Agents, function to offload tasks to quantum and classical worker nodes, agents.

Client can choose dynamic selection parameters like drop down populated based on user selections. And these can be returned to server as the configuration parameters. The data types will usually be pure classical code or pure quantum circuit based unitary operations, or can be hybrid code etc

Reference: https://arxiv.org/html/2506.01536v1

Key words: Klusch’s Quantum Computational Agents (QCA) model , Quantum Multi-Agent Reinforcement Learning (QMARL), Quantum Simulation for Decision-Making Agents, AI Agents for

Quantum Workflow Management, Automated Quantum Circuit Design and Optimization, Hamiltonian Variational Ansatz (HVA), inspired by the Quantum Approximate Optimization Algorithm (QAOA) and adiabatic quantum computation, Orchestration of Hybrid Quantum-Classical Systems, Figure 1. The anatomy of a Quantum Agent System Architecture, Agents to handle pure classical, pure quantum and hybrid tasks, Agents Using Quantum as a Subsystem, Maturity Model

Details

1) Quick literature-grounded observations (why this approach)

- The recent "Quantum Agents" paper frames the synergy two ways: (A) quantum-enhanced agent capabilities (e.g., better sampling/optimization/representation) and (B) agentic systems that orchestrate and improve quantum workflows. Use both directions. arXiv
- Practical systems in 2024–25 favor **hybrid quantum–classical** workflows where most control remains classical and short, targeted quantum subroutines are offloaded (QUBO, variational circuits, latent-space routines). Design agents to orchestrate short quantum tasks to limit decoherence and overhead. arXiv+1
- LLM/agentic coordination is maturing: multi-agent copilots for quantum sensing and code generation show the value of specialized agents (planner, verifier, quantum-coder, experimenter) cooperating with a knowledge base and active learning loops. Use agent specialization rather than one monolithic agent. arXiv

---

2) High-level architecture (onion → deployable container model)

Client layer (Thin & thick clients)

- Browser (React) primary; Desktop Win app later.
- Clients present: workflows, parameter dropdowns, job status, and dynamic selection of quantum vs classical offload.

Edge / Gateway

- API Gateway (REST/gRPC) + Auth (OIDC/JWT) + rate limiting.
- Client-side config schema validation; lightweight client agent (JS) to package requests.

Bhadale QAI Hub (Server / Orchestration)

- **Agent Orchestrator** (core service): receives tasks, decomposes into sub-agents, schedules workloads to classical or quantum worker pools.
- **Master Data & Library**: models, circuit templates, policy rules, LLM prompt templates, credentials. (Innermost onion).
- **Agent Registry**: list of agent types (Planner, Simulator, Optimizer, Verifier, QPU Broker, Data Curator, Safety Agent).
- **Workflow Engine**: stateful workflow/correlation (e.g., Temporal/Argo Workflows).
- **Job Queue**: support RPC, message queues (Kafka/RabbitMQ) for async steps.
- **Telemetry & Observability**: tracing, circuit execution logs, cost & latency metrics.

Worker pools

- **Classical Workers**: Python microservices (Flask/FastAPI), GPU/CPU-backed ML services.
- **Quantum Workers (QPU/Simulators)**: gateway to QPU providers (local QPU, cloud QPU via providers' SDKs). Implement a **QPU Broker** agent that manages queueing, approximate scheduling, and fallback strategies.
- **Hybrid Executors**: containers that host hybrid routines (e.g., PennyLane, Qiskit runtime, Cirq + classical pre/post-processing).

Security & Governance layer

- Identity, access, verification (ID verification agent), audit trails, policy enforcement engine (e.g., denylist for certain circuits), resource quotas.

Storage & Data

- Artifact store (model weights), circuit templates, result store (time series), secrets store (HashiCorp Vault).

Deployment

- Everything packaged as Docker containers, Helm charts + Kubernetes manifests for scaling, with sample K8s CRDs for Agent definitions.

---

3) Agent taxonomy (recommended) — core types & responsibilities

1. **Planner Agent** — decomposes high-level goals into tasks & sub-tasks, picks candidate agent pipeline.
2. **QPU Broker Agent** — translates circuit requests, handles provider selection, queuing, retries, cost/latency tradeoffs.
3. **Classical Executor Agent** — runs classical ML workloads and pre/post-processing.
4. **Hybrid Coordinator Agent** — orchestrates hybrid loops (e.g., VQE/variational loops: classical optimizer ↔ quantum evaluator).
5. **Verifier / Validator Agent** — semantic checks, syntactic quantum code tests, resource checks, safety checks.
6. **Data Curator Agent** — prepares datasets, compresses to latent spaces (autoencoders) when beneficial for QPU.
7. **Monitor / Safety Agent** — policy enforcement, anomaly detection, halting unsafe jobs.
8. **UI Assistant Agent** — client-facing LLM-based assistant to help with parameter selection and explainability.

Design principle: each agent is a small stateless container with a well-defined API. Stateful orchestration lives in the Workflow Engine.

---

4) Concrete artifacts to develop now (prioritized)

I. **Repo layout**

```
qai-agents/
  ├── hub/                        # Bhadale QAI Hub services
  ├── agents/                     # per-agent python packages
  │    ├── planner/
  │    ├── qpu_broker/
  │    └── verifier/
  ├── clients/
  │    ├── web/                    # React app
  │    └── win-desktop/            # later
  ├── infra/
  │    ├── k8s/                    # Helm charts, manifests
  │    ├── docker/                 # base dockerfiles
  │    └── ci/                     # CI pipelines
  └── docs/
```

II. **Environment file template** (`.env.example`)

```
# .env.example
HUB_API_URL=https://hub.example.com
AUTH_ISSUER=https://auth.example.com
JWT_AUDIENCE=bha-qai
QPU_PROVIDER_URL=https://qpu-provider.api
QPU_API_KEY=${QPU_API_KEY}
DB_URL=postgresql://qai_user:password@db:5432/qai
REDIS_URL=redis://redis:6379/0
SECRETS_VAULT_ADDR=https://vault:8200
ENABLE_SIMULATOR=true
WORKFLOW_ENGINE=temporal
```

III. **Kubernetes manifest (simple Job + Service for Planner agent)** — this scales to Helm later:

```
# infra/k8s/planner-deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: planner-agent
spec:
  replicas: 2
  selector:
    matchLabels:
      app: planner-agent
  template:
    metadata:
      labels:
        app: planner-agent
    spec:
      containers:
      - name: planner
        image: ghcr.io/yourorg/planner:0.1.0
        envFrom:
        - secretRef:
            name: qai-secrets
        - configMapRef:
            name: planner-config
        resources:
          limits:
            cpu: "2"
            memory: "4Gi"
---
apiVersion: v1
kind: Service
metadata:
```

```
  name: planner-service
spec:
  selector:
    app: planner-agent
  ports:
  - port: 8080
    targetPort: 8080
```

IV. **Example Python agent skeleton (FastAPI) — use this for most agents**

```
# agents/common/app.py
from fastapi import FastAPI, HTTPException
from pydantic import BaseModel

app = FastAPI(title="Planner Agent")

class PlanRequest(BaseModel):
    goal: str
    user_id: str
    constraints: dict = {}

class PlanResponse(BaseModel):
    plan_id: str
    steps: list

@app.post("/plan", response_model=PlanResponse)
async def plan(req: PlanRequest):
    # placeholder: decompose goal
    steps = [{"type":"analyze","params":{}},
{"type":"qpu_call","params":{"circuit":"..."} }]
    return PlanResponse(plan_id="plan_12345", steps=steps)
```

V. **Agent registration CRD (conceptual)**

```
apiVersion: qai.example.com/v1
kind: Agent
metadata:
  name: qpu-broker
spec:
  image: ghcr.io/yourorg/qpu-broker:0.1.0
  capabilities: ["qpu_call","schedule","fallback"]
  resources:
    cpu: "2"
    memory: "8Gi"
```

VI. **Client UI design notes (React)**

- Dynamic parameter dropdowns come from `GET /hub/parameter-metadata?workflow=abc`.
- Client sends a single JSON job descriptor:

```
{
  "workflow":"molecule_optim",
  "params":{"ansatz":"UCCSD","shots":4096,"use_qpu":true},
  "user_id":"vijay",
  "auth_token":"..."
}
```

- Provide progress stream via WebSocket or server-sent events for low-latency updates.

VII. **Config schema (JSON Schema)**

- Create a canonical JSON Schema for job descriptor, circuit descriptor (type: "quantum" | "classical" | "hybrid"), resource hints, cost/latency sensitivity.

---

5) Safety, governance, and lifecycle guardrails

- **Policy agent**: enforce allowed gates for circuits (e.g., no untrusted entanglement requests, deny blacklisted operations).
- **Explainability**: every plan must include a provenance block: which agent produced the step, inputs, model versions.
- **Human-in-loop thresholds**: define cost/latency/sensitivity thresholds that require manual approval.
- **Testing harness**: unit tests for classical parts; circuit emulation tests using local simulators; regression tests for hybrid loops.
- **Data protection**: all QPU job payloads should be scanned for sensitive PII & optionally redacted; use encrypted-at-rest artifact store and access control via Vault.
- **Observability for quantum runs**: capture circuit transpilation info, QPU metadata (backend name, noise profile, queue wait), and result samplings.

---

6) Development & CI suggestions

- Keep a canonical `qai-runtime` Python package used by all agents (auth, tracing, telemetry, retry logic).
- Use GitHub Actions/ GitLab CI to build & publish images.
- Use unit tests + integration tests that mock QPU providers (and real QPU smoke tests gated as optional CI jobs).
- Provide a developer-mode `docker-compose` stack that runs Hub + Planner + QPU-simulator for local dev.

---

7) Example lifecycle mapping (product → deployed agent)

1. Idea → spec (workflow + param schema)
2. Prototype → simple Planner + one QPU Broker + local simulator
3. Implementation → full hybrid coordinator + verifier + production QPU Broker
4. Integration → connect to Bhadale Hub master data, Auth, Vault
5. Testing → emulation + real QPU runs (staged)
6. Deployment → Helm chart + autoscaling policies
7. Monitoring & MLOps → model/circuit drift detection, retraining strategies

---

8) Minimal PoC to build first (priority)

- Planner Agent (FastAPI)
- QPU Broker (calls a simulator like Qiskit Aer / PennyLane local backend)
- Simple web client (React) that lets user pick a workflow, fill params, submit, and watch status.
- K8s manifests + Helm for those three services.
  This PoC shows end-to-end: UI → Hub/Planner → QPU Broker → Simulator → results.

---

9) Useful tech stack choices (recommended)

- Python: FastAPI, Pydantic, asyncio (agents)
- Quantum SDKs: PennyLane (good hybrid support) and Qiskit (broad provider ecosystem)
- Workflow: Temporal or Argo Workflows (Temporal preferred for complex stateful orchestrations)
- Messaging: Kafka if high throughput; RabbitMQ for simpler queues
- Containers/orchestration: Docker + Kubernetes + Helm
- Secrets: HashiCorp Vault
- Observability: OpenTelemetry, Prometheus + Grafana
- Frontend: React + TypeScript; websockets for streaming status

---

10) Short list of deliverables I can produce for you next (pick any)

- Full **PoC repo skeleton** (files shown above + README + basic CI)
- Detailed **Helm chart** and production-grade K8s manifests with HPA & resource requests
- A **Planner + QPU Broker** working FastAPI duo (Python code) + Dockerfiles (no live QPU access; uses simulator)
- React sample client that connects to the Planner and streams progress (built with create-react-app)
- JSON Schema for the job descriptor and example job payloads
- Security & governance checklist and policy templates (audit rules, human-approval thresholds)

//

1) High-level pattern — how it all fits

- **QAI Execution Pipeline (E-Pipeline)** — central, deterministic flow that takes a high-level client job → decomposes it → issues sub-tasks to agents/workers → aggregates & post-processes results → returns a single canonical response. It's the "control plane" for an end-to-end job and lives in the Hub (Workflow Engine + Agent Orchestrator).
- **Multi-Agent Manager ("Crew")** — library/service inside the Hub that maintains agent lifecycle, capabilities, health, and matchmaking (which agent(s) should handle a step). Treat as a registry + scheduler + federation manager.
- **Middleware** — lightweight, well-defined plumbing between client/hub/workers: API Gateway (gRPC/HTTP), message bus (Kafka/RabbitMQ), RPC layer (gRPC, Celery for Python tasks), and event

streaming (SSE/WebSockets). Middleware also enforces serialization formats (JSON Schema, Avro) for job descriptors, circuit descriptors, telemetry.

- **Bhadale QAI Hub** — the integration hub and single source of truth: master data, policy engine, artifact store, credentials, capability registry, telemetry, and policy enforcement. All agents register here and the E-Pipeline consults it during planning and runtime.
- **Dependency & Hardware Resolution** — an **Adapter/Driver** layer that abstracts QPU providers, simulators, quantum SDKs (Qiskit, PennyLane, Cirq), cloud runtimes (Azure Quantum), and internal QAI Processor/Datacenter drivers. The QPU Broker uses that layer to choose provider + runtime + backend, and to translate/compile circuits to provider-specific formats.

2) Concrete components & responsibilities

A. QAI Execution Pipeline (E-Pipeline)

- **Job Intake**: validate job descriptor (JSON Schema), auth, quota check.
- **Planner**: decomposes job into steps and selects candidate agent pipelines (uses Crew).
- **Splitter**: splits hybrid jobs into classical tasks, short/near-term quantum tasks (minimize decoherence), and long running experiments.
- **Scheduler**: map steps to worker pools and runtimes (classical/hybrid/QPU). Enforces policy/human approval thresholds.
- **Executor**: issues RPCs / messages to agents and tracks state (Temporal/Argo for durable workflows).
- **Aggregator / Recomposer**: reassembles subresults, runs cross-verification, computes provenance & final artifacts.
- **Result Publisher**: stores results, telemetry, provenance, notifies client.

B. Crew (Multi-Agent Manager)

- Capability registry with metadata: capabilities, supported runtimes, SLAs, trust level, cost profile.
- Health & heartbeat service; k8s-aware redeployment strategy.
- Dynamic discovery (service mesh + sidecar) for autoscaling and location-aware routing.
- Policy tags (e.g., “trusted”, “experimental”, “simulator_only”) so planner picks appropriate agents.

C. Middleware & Integration

- **API Gateway**: Auth, rate-limiting, input validation, A/B routing.
- **Message Bus**: Kafka for high throughput; RabbitMQ/Celery for task orchestration; fallback to direct gRPC for small requests.
- **RPC/Adapters**: gRPC for agent → agent communications (typed protobufs), REST for client → hub.
- **Data Plane**: artifact store (S3), results DB (time series + document store), telemetry pipeline (OTel → Prometheus → Grafana).

D. Adapter / Driver Layer (Dependency resolution)

- Plugin-based adapter model:
  - `adapters/qiskit_adapter` — translates to Qiskit circuit objects, Qiskit Runtime calls (or Aer for simulator).
  - `adapters/azure_quantum_adapter` — translates to Azure job format, uses Azure SDK + provider-specific topologies.
  - `adapters/internal_qai_processor_adapter` — low-level driver for your QAI Processor (binary/IDL bindings or REST/gRPC shim).

    - `adapters/simulator_adapter` — local simulators (PennyLane, Qiskit Aer, Cirq).
- Each adapter exposes capability descriptors (shots, noise_model_support, mid-circuit-measurements, topology) and a `compile()` + `submit()` + `query()` surface.
- Adapter registry in Hub selects best candidate based on job constraints (latency, cost, fidelity, availability, policy).

E. Use of your internal assets (QAI OS, Processor, Datacenter, Frameworks)

- Treat internal assets as first-class provider(s): implement adapters and a **local QPU Broker policy** that can prefer internal hardware when:
    1. internal capacity available,
    2. the job's sensitivity requires on-prem, or
    3. cost/latency tradeoffs favor it.
- QAI OS: deploy a **runtime manager** inside the Hub that can orchestrate local QAI OS images (VMs/containers) on the QAI Datacenter; use the QAI Processor for tightly coupled workloads via a low-latency gRPC interface or PCIe / shared-memory shim as appropriate.
- QAI Datacenter: integrate with scheduler (Slurm/Kubernetes) and telemetry to allocate nodes.
- PLM / Org / Enterprise frameworks: register transformations, approvals, IP tags, and lifecycle metadata into Hub artifacts so product lifecycle and governance are attached to every job and agent artifact.

3) Runtimes & SDK support strategy

- **Runtime abstraction**: define `RuntimeCapability` interface: supports (shots, runtime_execution, param_sweep, QASM, mid-circuit, dynamic circuits, noise_model). Each adapter implements it.
- **Supported runtimes initially**:
    - Qiskit Runtimes (Qiskit Runtime/RuntimeService pattern)
    - PennyLane / Pennylane-Lightning for hybrid variational workflows
    - Cirq for hardware with native topology
    - Azure Quantum runtimes via Azure SDK + provider-specific backends (IonQ, Honeywell/Quantinuum) — adapter handles submission and result normalization
    - Internal runtime (QAI OS/Processor runtime)
- **Fallback & simulation policy**: if requested runtime is unavailable, automatically fallback to simulator or queued execution, with user-visible status & human approval when needed.

4) Software engineering & systems engineering split

- **Software engineering (QAI agents & middleware)**:
    - Python microservices (FastAPI) for agents, standard `qai-runtime` shared package (auth, retry, telemetry).
    - Unit tests + adapter mocks + integration tests with local simulators.
    - CI pipelines that build & push Docker images, run smoke tests, and optionally run gated QPU tests in staging.
- **Systems engineering (hardware platforms)**:
    - Driver integration with QAI Processor (firmware/interface requirements), kernel/driver support for any specialized hardware.
    - Capacity planning for QAI Datacenter (cooling, noise, isolation), network latencies, and fault domains.
    - Heterogeneous scheduling policies for classical-only vs hybrid jobs.
    - Security: HSM / TPM integration, vaulting keys for hardware access.

- Cross-discipline: define interface contracts (OpenAPI + gRPC protobufs) to clearly separate responsibilities.

5) Governance, PLM & Org framework integration

- **PLM hooks**: On spec creation, create PLM artifact (versioned). Every agent artifact (code, circuit, model) must include PLM metadata: owner, version, maturity, approved_by.
- **Approval workflows**: human-in-loop approvals wired into the E-Pipeline via the Workflow Engine for sensitive/expensive runs.
- **Org framework**: map teams/roles to agent ownership and escalation; RBAC in the Hub enforces role-based policies.
- **Audit & compliance**: immutable logs of job manifests, circuits submitted, provider used, and result fingerprints. Integrate with SIEM.

6) Testing, verification & validation

- **Unit & CI**: adapter mocks, agent contract tests.
- **Simulation & emulation**: full pipeline with local simulators (Qiskit Aer / PennyLane Lightning).
- **Staged QPU runs**: smoke tests against selected QPU backends in staging with cost caps.
- **Hybrid loop regression**: automated tests for variational circuits (VQE / QAOA) to watch optimizer convergence patterns.
- **Property-based testing**: ensure circuit transpilation preserves intended properties (e.g., measurement statistics within expected variance).

7) Practical operational considerations

- **Cost & quota management**: per-user and per-project quotas on QPU time; billing metadata added to job provenance.
- **Provenance & explainability**: attach versioned models, circuit transpiler options, noise model used, provider details to results.
- **Observability**: per-job traces (distributed tracing), metric dashboards (latency, queue length, success rate), alerting for drift and anomalies.
- **Security**: circuit content scanning for restricted constructs, secret-sanitization, encrypted artifacts.

8) Implementation roadmap (pragmatic)

Phase 0: design & adapters

- Define job schema, adapter API, Planner/Crew interfaces.
- Provide Qiskit & simulator adapters + a local QAI Processor shim.

Phase 1: PoC E-Pipeline

- Planner + Crew + QPU Broker + Simulator + React UI.
- Basic k8s manifests + docker compose dev stack.

Phase 2: Integrations

- Add Azure Quantum adapter, internal datacenter scheduler, metadata/PLM integration.

Phase 3: Harden + Policies

- Policy agent, Vault integration, monitoring, cost controls, HA.

//

QAI_Agents_Framework — Text Block Diagram (modules · functions · I/O · product stack · QAI merit · lifecycle · dependencies)

Below is a compact textual block diagram of the full framework, followed by a per-block mapping of **Functions | Input → Output | Product stack examples | QAI merit | Lifecycle phase | Key dependencies (runtimes, libs, infra)**.

```
[ CLIENTS ]
  ├─ Browser UI (React)  <--->  [API Gateway / Edge]
  └─ Win Desktop / Mobile       (Auth, Validation, SSE/WebSocket)

                          ||
                          \/
[ API GATEWAY / EDGE ] (REST/gRPC, OIDC/JWT, RateLimit, Validation)
                          ||
                          \/
[ BHADALE QAI HUB (Control Plane) ]
  ├─ Intake / Auth / Validation
  ├─ Job Registry / PLM / Master Data
  ├─ Crew (Multi-Agent Manager / Registry)
  │     ├─ Agent Catalogue (planner, qpu_broker, verifier, data_curator, monitor,
ui_assistant, hybrid_coordinator)
  │     └─ Health / Capability / Matchmaking
  ├─ E-Pipeline (Execution Pipeline / Workflow Engine)
  │     ├─ Planner    (decompose job → pipeline)
  │     ├─ Splitter   (classical / quantum / hybrid)
  │     ├─ Scheduler  (match to worker pools & runtimes)
  │     ├─ Executor   (issue tasks to agents, track state)
  │     └─ Aggregator (re-compose subresults, provenance)
  ├─ Policy & Governance Agent (policy engine, approvals)
  ├─ Adapter Registry (adapters: qiskit, pennylane, cirq, azure, internal_qai)
  ├─ Artifact Store / Vault / Telemetry / Observability
  └─ API to Clients (status, results, provenance)

                          ||
                          \/
[ MIDDLEWARE / MESSAGE BUS ]
  ├─ gRPC / REST / WebSockets
  ├─ Kafka / RabbitMQ / Temporal (durable workflows)
  └─ Service Mesh (Istio/Linkerd)

                          ||
                          \/
[ WORKER POOLS (Execution Plane) ]
  ├─ Classical Workers (python microservices, GPUs)
  ├─ Hybrid Executors (PennyLane, Pennylane-Lightning containers)
  ├─ QPU Broker Agents  ---> Adapter Layer
  │        ├─ Qiskit Runtime (Aer / IBM Q / Runtime)
  │        ├─ Azure Quantum Runtime (IonQ / Quantinuum etc.)
  │        ├─ Cirq Backends / Hardware-Specific Drivers
```

```
  │          └─ Internal QAI Processor / QAI OS runtime (on-prem)
  └─ Simulators (Qiskit Aer / statevector, noisy simulators)

                          ||
                          \/
[ INFRA / DEPLOYMENT ]
  ├─ Kubernetes + Helm (CRDs for Agent definitions)
  ├─ CI/CD (GitHub Actions / GitLab)
  ├─ Vault, Postgres, Redis, S3-compatible storage
  └─ QAI Datacenter / On-prem hardware (QAI Processor nodes)

                          ||
                          \/
[ MONITORING / AUDIT / PLM / ORG INTEGRATION ]
  ├─ Telemetry (OpenTelemetry → Prometheus → Grafana)
  ├─ Audit Logs / SIEM
  └─ PLM hooks (artifact lifecycle, approvals, owners)
```

Block details — each block mapped to: Functions | Input → Output | Product Stack | QAI merit | Lifecycle phase | Dependencies

1) CLIENTS (Browser / Desktop / Mobile)

- Functions: UI for job composition, parameter selection, human approvals, live status & visualization.
- Input → Output: user job descriptor (JSON) → job submission; displays results, provenance, visual charts.
- Product stack: React + TypeScript, WebSocket/SSE, Electron (for Win desktop).
- QAI merit: enables human-in-loop; UX for selecting hybrid vs QPU options; explainability.
- Lifecycle phase: Idea → Prototype → Validation → Production.
- Dependencies: API Gateway endpoints, Auth (OIDC/JWT), schema (JSON Schema).

---

2) API GATEWAY / EDGE

- Functions: auth, validation, rate-limiting, routing to Hub, schema enforcement, initial quota check.
- Input → Output: HTTP/gRPC requests → validated job envelope (forwarded to Hub).
- Product stack: Kong/Envoy/NGINX, OIDC provider, OpenAPI.
- QAI merit: ensures secure & deterministic intake for QPU-sensitive jobs.
- Lifecycle phase: Design → Harden → Operate.
- Dependencies: Identity provider, TLS, Schema registry.

---

3) BHADALE QAI HUB (Control Plane)

- Functions: central coordinator — Job intake, Crew, E-Pipeline, Policy, Adapter Registry, PLM, Artifact store.
- Input → Output: validated job → decomposed tasks & final aggregated result.
- Product stack: FastAPI (services), Temporal/Argo, Postgres, Redis, Vault, S3, PLM system integration.
- QAI merit: Orchestrates hybrid decomposition, optimizes QPU offload, ensures governance & provenance.

- Lifecycle phase: All phases (core platform).
- Dependencies: Workflow engine, adapters, policy engine, telemetry.

Subcomponents:

- Crew (Agent Manager): capability discovery, matchmaking. Depends on service mesh, k8s.
- E-Pipeline: planner, splitter, scheduler, executor, aggregator. Depends on Temporal/Argo, message bus.
- Adapter Registry: plugin system for Qiskit, Azure, Pennylane, Internal QAI. Depends on SDKs and adapter implementations.

---

4) MIDDLEWARE / MESSAGE BUS

- Functions: reliable messaging, RPC, streaming of job events; durable task passing.
- Input → Output: tasks/events ↔ worker agents.
- Product stack: Kafka (events), RabbitMQ/Celery (tasks), gRPC, WebSockets, Istio.
- QAI merit: supports low-latency streaming for interactive quantum loops; decouples components.
- Lifecycle phase: Prototype → Scale.
- Dependencies: Network, serializers (Avro/JSON), schema registry.

---

5) WORKER POOLS (Execution Plane)

- Functions: execute classical ML, run hybrid variational loops, submit/monitor QPU jobs, simulate circuits.
- Input → Output: task (classical/quantum descriptor) → result samples, metrics, logs.
- Product stack:
  - Classical: Python (FastAPI), PyTorch/TensorFlow.
  - Hybrid: PennyLane, Pennylane-Lightning, JAX.
  - Quantum runtimes: Qiskit (Runtime/Aer), Cirq, Azure Quantum SDK, Internal QAI runtime.
- QAI merit: runs short quantum subroutines (VQE, QAOA, sampling), enables classical optimization loops, speeds up search/optimization via quantum primitives where beneficial.
- Lifecycle phase: Prototype → Integration → Production.
- Dependencies: Quantum SDKs (qiskit, pennylane, cirq), provider credentials (IBM/Azure/etc.), internal QAI OS driver APIs.

Key special block — **QPU Broker Agent**

- Functions: selects runtime/provider, compiles/transpiles circuits, submits jobs, handles retries & fallbacks.
- QAI merit: abstracts hardware heterogeneity and optimizes for fidelity/latency/cost tradeoffs.
- Dependencies: Adapter implementations for each runtime, provider SDKs.

---

6) ADAPTER LAYER (Qiskit / Azure / Internal / Simulator)

- Functions: translate canonical circuit/job descriptor into provider-specific calls; normalize results.
- Input → Output: canonical circuit descriptor → provider-specific job object & normalized result.

- Product stack: adapter packages (adapters/qiskit_adapter, adapters/azure_adapter, adapters/internal_qai_adapter).
- QAI merit: portability — same high-level workflow can use different quantum backends transparently.
- Lifecycle phase: Design → Implement → Maintain.
- Dependencies: qiskit, qiskit-ibm-runtime, azure-quantum-sdk, pennylane, cirq, internal SDKs/drivers.

---

7) SIMULATORS

- Functions: emulate circuits for development & verification; act as fallback when QPU unavailable.
- Input → Output: circuit → samples/statevector/noisy outputs.
- Product stack: Qiskit Aer, Pennylane Lightning, Qulacs (optional).
- QAI merit: crucial for testing, regression and for short-circuiting expensive QPU runs.
- Lifecycle phase: Dev & Test; used in Staging.
- Dependencies: qiskit-aer, pennylane, compute resources (CPU/GPU).

---

8) INFRA / DEPLOYMENT (Kubernetes, Helm, Datacenter)

- Functions: run containers, autoscale agents, manage secrets, schedule heavy jobs to QAI Datacenter.
- Input → Output: container images & Helm charts → deployed services & nodes.
- Product stack: Kubernetes, Helm, CRDs for Agent, Flux/ArgoCD, GitOps CI.
- QAI merit: ensures reproducible, scalable deployments of heterogeneous components including on-prem QAI Processors.
- Lifecycle phase: Deploy → Operate.
- Dependencies: Docker registry, CI pipeline, on-prem networking, datacenter scheduler.

---

9) MONITORING / PLM / ORG INTEGRATION

- Functions: telemetry, auditing, PLM lifecycle hooks, RBAC mapping to Org framework, cost & quota tracking.
- Input → Output: job telemetry & artifacts → dashboards, audit trails, PLM records.
- Product stack: OpenTelemetry, Prometheus, Grafana, ELK, SIEM, PLM system.
- QAI merit: traceability, reproducibility, compliance for QAI experiments and products.
- Lifecycle phase: Operate → Govern → Retire.
- Dependencies: logging/metrics libs, PLM API, SIEM, Vault.

Quick legend of "QAI merit" terms used above

- **Orchestration merit**: ability to decompose tasks and place most valuable quantum workloads properly.
- **Hybrid acceleration**: speed/quality improvements by combining classical optimizers with quantum evaluations.
- **Portability**: using adapters to run same workload on variety of runtimes with minimal change.
- **Governed experimentation**: safe, auditable experimentation lifecycle for quantum workloads in enterprise settings.

- **Explainability & Provenance**: tracking planner decisions, agent versions, provider metadata.

One-line canonical job flow (summary)

1. Client → API Gateway (validated job descriptor)
2. Hub Intake → Planner (decomposes) → Crew matches agents
3. E-Pipeline Scheduler assigns to Worker Pools (Classical / Hybrid / QPU Broker)
4. QPU Broker → Adapter → Provider runtime (Qiskit / Azure / Internal) or Simulator
5. Results → Aggregator (post-process, verify) → Persist in Artifact Store + PLM → Notify Client

//

Executive summary of changes (one-paragraph)

We extend the Hub and Worker design with new agent classes and subsystems: (1) adopt a **Quantum Computational Agent (QCA)** pattern for agents that embed quantum processing primitives inside the agent body; (2) add explicit support for **Quantum Multi-Agent Reinforcement Learning (QMARL)** training and deployment pipelines; (3) provide an integrated **Quantum Simulation Sandbox** for decision-making and planning; (4) add an **Automated Circuit Design & Optimizer** service (with HVA / QAOA templates and a learnable circuit generator); (5) expand the **QPU Broker / Adapter** with topology-aware transpilation and hybrid orchestration for ansatz strategies (HVA/QAOA); and (6) integrate a **Maturity Model** stage metadata into the PLM and Planner to guide allowed runtime and approval workflows. These changes preserve the onion structure while making quantum-specific capabilities first-class. arXiv

---

Detailed updates mapped to the existing architecture

1. New agent types and extensions

Add these agent types (or extend existing ones) in the **Crew / Agent Catalogue**:

- **Quantum Computational Agent (QCA)**
  - Role: an agent whose *internal* computation can directly include quantum subroutines (the agent's decision function can call a local quantum routine or QPU via the Adapter). Follows the design concept of Klusch's QCA (master-slave style hybrid agent).
  - Where: implemented as an agent image (container) with `qai-runtime` + quantum SDK adapters and a probabilistic decision module. Marked with capability tag `qca`. arXiv
- **QMARL Trainer & Executor Agents**
  - Role: centralized trainer that runs multi-agent reinforcement learning experiments (centralized training, decentralized execution), manages experience replay (or quantum memory), curriculum schedule, reward shaping, and deploys policy circuits to QCA agents.
  - Where: sits in Hub (training pipeline) and worker pools (GPU + QPU hybrid runners). Log metadata into PLM. arXiv
- **Simulation Agent / Quantum Sandbox**

    - Role: provide fast parallel quantum simulations and scenario sampling for decision-making agents (internal "mental model" of environment). Exposes API for planners and agents to run many simulated world-rollouts.
    - Where: Worker pool (simulator cluster) + E-Pipeline call for planning loops. Useful for agents needing internal model-based planning. arXiv
- **AutoCircuit Designer & Optimizer Agent**
    - Role: Given a high-level specification (objective / cost / gates constraints), synthesize candidate circuits (including HVA / QAOA ansatz families), run quick evaluations (simulator or small QPU runs), and propose optimized circuits. Integrates differentiable ansatz tools and search/ML-based generators.
    - Where: Hub's Adapter Registry & Execution Pipeline call this agent as part of the "compile/optimize" step. arXiv
- **Hybrid Orchestrator Agent (extended Hybrid Coordinator)**
    - Role: Manages classical optimizer ↔ quantum evaluator loops (VQE, QAOA) including mid-circuit calls, early-stop, and fidelity-aware scheduling. Interfaces tightly with QPU Broker & Adapter for topology-aware transpilation. arXiv
- **Agent-level Verifier / Safety Agent (quantum-aware)**
    - Role: semantic circuit checks (topology violations, forbidden gates), resource constraints (shots, depth), and risk scoring (sensitivity, IP). Trigger approval flows for immature jobs as per Maturity Model.

---

2. Planner / E-Pipeline changes (how decomposition & reassembly change)

- **Planner**: enhance planner heuristics to:
    - Recognize when tasks are amenable to QCA or QMARL (e.g., combinatorial optimization, stochastic simulation).
    - Request *simulation rollouts* from the Simulation Agent to estimate expected value of quantum offload. arXiv
- **Splitter**: classify subtasks into: `classical`, `quantum_subroutine` (short, low depth), `quantum_simulation` (parallelizable), and `qca_task` (agent internal quantum call). This helps select adapter and approval path.
- **Scheduler**: use a policy that factors:
    - Maturity stage (from PLM metadata) → whether a job may use production QPU or only simulator / staging QPU.
    - QMARL experimental runs get special trainer queues and sampling budgets.
    - Hybrid loop scheduling: prioritize low-latency runtimes or runtime services (e.g., Qiskit Runtime) for tight optimizer loops.
- **Aggregator**: when recomposing results from QMARL or QCA-powered agents, collect provenance: which agent's policy circuit was used, training checkpoint ID, simulator/noisy-QPU characteristics, and uncertainty measures. Store these in the Artifact Store & PLM.

---

3. QPU Broker & Adapter enhancements (topology-aware & ansatz-aware)

- Adapters must expose `capabilities` including support for:
    - mid-circuit measurement, parameterized circuits, dynamic circuits (runtime), native topology, noise model metadata, and runtime queue latency.
- QPU Broker must implement:

- **Topology-aware transpilation**: select or transform ansatz (e.g., HVA/QAOA) to fit the backend topology while preserving problem structure.
- **Ansatz library**: maintain HVA/QAOA templates parameterized for common problem families; the AutoCircuit agent can instantiate/modify these. arXiv

---

4. QMARL & Training pipeline (ML infra)

- Add a **QMARL Training Service** in Hub:
  - Centralized trainer orchestrates many episodes (simulator + QPU as needed), aggregates gradients or policy updates, and snapshots policy circuits to Artifact Store/PLM.
  - Supports *decentralized execution*: policies are pushed to QCA agents; they run with local inference + occasional policy updates. arXiv
- Data flows: experience → trainer → policy circuit → PLM → deployment.

---

5. Simulation sandbox & decision-making

- Provide API for agents/planners to request probabilistic rollouts across many parallel simulated trajectories using quantum simulators (or classical approximations when suitable). Use batched/sampled execution to let planners get expected-utility estimates quickly. This is particularly valuable for agents in chemistry/materials or quantum-control domains. arXiv

---

6. Automated Circuit Design & HVA / QAOA integration

- Add **AutoCircuit Designer** microservice that:
  - Accepts cost function + constraints + target backend capabilities.
  - Proposes HVA/QAOA-style ansatz or learned circuit via generative models (LLM or GNN) plus gradient-free/genetic search.
  - Runs quick simulations or low-shot QPU evaluations (budgeted) and returns ranked circuits with expected fidelity & cost. arXiv
- This service integrates with Planner (compile phase) and Adapter (to produce provider-specific circuits).

---

7. Operational modes & agents using quantum as subsystem

- Explicitly support the two operational modes described in the paper:
  1. **Quantum-assisted agency**: agents use quantum subroutines internally (QCA agents). Planner treats these as `qca_task`. arXiv
  2. **Quantum-centric control**: when agent's decision substrate is quantum (rare but possible), the Hub must provide specialized runtime and more stringent governance (higher maturity required).
- Include a hybrid/blurred mode where agent control is split across classical and quantum layers.

---

8. Maturity Model mapped to PLM & Workflow Engine

Integrate a **Maturity Stage** field into every workflow/job descriptor and PLM artifact. Typical stages (increasing trust & allowed runtime privileges):

- **Concept / Idea** — allowed: local dev simulators only.
- **Prototype** — allowed: staging QPU (low-cost, limited shots). Human-in-loop required for high-cost runs.
- **Validated** — allowed: production QPU with quota & automated monitoring.
- **Certified** — allowed: production QPU with elevated quotas, SLA. Requires governance approvals, security review.
  Map these to Planner & Scheduler policies: a job's maturity stage controls which adapters/backends and agent capabilities can be used. This directly realizes the paper's maturity model recommendations. arXiv

---

9. Figure-1 Anatomy (paper) — where to reflect it in our docs

The paper's "anatomy of a Quantum Agent System Architecture" suggests modules for Perception, Memory, Decision, and Effectors; map those to our Hub & QCA design:

- **Perception** → Data Curator Agent & sensors (classical inputs).
- **Memory** → Artifact Store + Quantum Memory primitives (where available) used by QCA & QMARL.
- **Decision** → Planner + QCA / QMARL agents.
- **Effectors** → Worker Pools, QPU Broker, external actuators.
  Include a "Figure-1-style" diagram in your docs and cross-link module responsibilities. arXiv

---

10. Policies, testing & governance additions

- **Experiment budgets** and **maturity enforcement** enforced by Policy Agent. QMARL training jobs require separate quotas and staging-only flags until validated.
- **Evaluation suites**: for QMARL and QCA policies, add standardized benchmarks (learning curves, sample complexity, stability under noise).
- **Interoperability**: require adapter conformance tests (preserve statistics under transpilation).
- **Provenance**: log training seed, circuit ansatz version, noise model metadata, and adapter transformations.

Concrete implementation checklist (what to create next)

If you want I'll generate these artifacts right now (pick any, I can produce immediately in this session):

1. **Updated Agent Catalogue YAML** (adds QCA, QMARL, AutoCircuit types with capabilities and maturity tags).
2. **Planner heuristics doc** (pseudocode) showing detection rules for when to delegate to QCA vs classical agent and how to call Simulation sandbox for rollouts.

3. **AutoCircuit agent skeleton (FastAPI)** that accepts objective + backend constraints and returns candidate HVA/QAOA circuits (stubs).
4. **QMARL training pipeline README** (architecture + job spec example).
5. **Maturity model policy JSON** and a sample enforcement hook for Scheduler (code snippet).
6. **Figure-1 style mermaid diagram** of the Quantum Agent anatomy mapped to our Hub (visual).

//

Updated QAI_Agents_Framework — Text Block Diagram (with QCA, QMARL, AutoCircuit, Simulation Sandbox, Maturity Model)

Below is the **updated compact textual block diagram** that incorporates the paper-derived concepts (Klusch's Quantum Computational Agents, QMARL, Simulation Sandbox, Automated Circuit Design with HVA/QAOA support), plus a per-block mapping of **Functions | Input → Output | Product stack examples | QAI merit | Lifecycle phase | Key dependencies**.

```
[ CLIENTS ]
  ├─ Browser UI (React)  <--->  [API Gateway / Edge]
  └─ Win Desktop / Mobile       (Auth, Validation, SSE/WebSocket)

                          ||
                          \/
[ API GATEWAY / EDGE ] (REST/gRPC, OIDC/JWT, RateLimit, Validation)
                          ||
                          \/
[ BHADALE QAI HUB (Control Plane) ]
  ├─ Intake / Auth / Validation
  ├─ Job Registry / PLM / Master Data  (includes Maturity Stage metadata)
  ├─ Crew (Multi-Agent Manager / Registry)
  │     ├─ Agent Catalogue (planner, qca, qmarl_trainer, auto_circuit, qpu_broker,
verifier, data_curator, monitor, ui_assistant, hybrid_coordinator)
  │     └─ Health / Capability / Matchmaking (capability tags, maturity policies)
  ├─ E-Pipeline (Execution Pipeline / Workflow Engine)
  │     ├─ Planner    (decompose job → pipeline; maturity-aware heuristics)
  │     ├─ Splitter   (classical / quantum_subroutine / qca_task / simulation_task)
  │     ├─ Scheduler  (match to worker pools & runtimes; enforce maturity & quotas)
  │     ├─ Executor   (issue tasks to agents, track state)
  │     └─ Aggregator (re-compose subresults, verify, provenance)
  ├─ Policy & Governance Agent (maturity enforcement, approvals, budgets)
  ├─ Adapter Registry (adapters: qiskit, pennylane, cirq, azure, internal_qai) —
topology-aware & ansatz-aware descriptors
  ├─ AutoCircuit Designer & Optimizer (HVA / QAOA templates + learned generator)
  ├─ QMARL Training Service (centralized trainer → policy artifacts)
  ├─ Quantum Simulation Sandbox (batched rollouts for planners/agents)
  ├─ Artifact Store / Vault / Telemetry / Observability / PLM hooks
  └─ API to Clients (status, results, provenance, explainability)

                          ||
                          \/
[ MIDDLEWARE / MESSAGE BUS ]
  ├─ gRPC / REST / WebSockets
  ├─ Kafka / RabbitMQ / Temporal (durable workflows)
  └─ Service Mesh (Istio/Linkerd)

                          ||
                          \/
[ WORKER POOLS (Execution Plane) ]
  ├─ Classical Workers (python microservices, GPUs)
```

```
  ├─ Hybrid Executors (PennyLane, Pennylane-Lightning, JAX)
  ├─ QCA Agents (containers embedding quantum subroutines)
  ├─ QMARL Agents (deployed policy circuits running locally / at edge)
  ├─ AutoCircuit Workers (synthesis & evaluate candidate circuits)
  ├─ QPU Broker Agents  ---> Adapter Layer
  │          ├─ Qiskit Runtime (Aer / IBM Q / Runtime)
  │          ├─ Azure Quantum Runtime (IonQ / Quantinuum etc.)
  │          ├─ Cirq Backends / Hardware-Specific Drivers
  │          └─ Internal QAI Processor / QAI OS runtime (on-prem)
  └─ Simulators (Qiskit Aer / statevector, noisy simulators in sandbox)

                        ||
                        \/
[ INFRA / DEPLOYMENT ]
  ├─ Kubernetes + Helm (CRDs for Agent definitions, maturity labels)
  ├─ CI/CD (GitHub Actions / GitLab)
  ├─ Vault, Postgres, Redis, S3-compatible storage
  └─ QAI Datacenter / On-prem hardware (QAI Processor nodes)

                        ||
                        \/
[ MONITORING / AUDIT / PLM / ORG INTEGRATION ]
  ├─ Telemetry (OpenTelemetry → Prometheus → Grafana)
  ├─ Audit Logs / SIEM
  ├─ PLM hooks (artifact lifecycle, approvals, owners, maturity)
  └─ Evaluation Benchmarks (QMARL, QCA performance & fidelity)
```

Per-block updates & notes (new items emphasized)

CLIENTS

- Functions: job composition, selecting **maturity stage** (or default), request simulator vs production QPU, approvals UI for sensitive runs.
- QAI merit: enables human oversight for experiments that involve QCA/QMARL assets.
- Dependencies: API Gateways, Auth, JSON Schema that includes maturity and mission tags.

API GATEWAY / EDGE

- Add validation rules for Maturity Stage field; route staging QPU runs to sandbox endpoints automatically.

BHADALE QAI HUB (Control Plane) — key new subcomponents

- **Job Registry / PLM**: store maturity stage and PLM metadata; artifact versioning for policy circuits and ansatz templates.
- **Crew / Agent Catalogue**: now includes `qca`, `qmarl_trainer`, `auto_circuit`, and tags for `maturity` and `runtime_capabilities`.
- **Planner**: maturity-aware heuristics — consults Simulation Sandbox to estimate EV (expected value) before QPU offload.
- **Splitter**: recognizes `qca_task` (agent-internal quantum call) vs `quantum_subroutine`.

- **Scheduler**: enforces maturity policy: Concept→Simulator-only, Prototype→staging QPU, Validated→limited production QPU, Certified→full production QPU.
- **Policy & Governance Agent**: budget & quota enforcement for QMARL training; human-in-loop gating for low-maturity high-cost runs.
- **AutoCircuit Designer & Optimizer**: HVA/QAOA templates, learned circuit generator, ranking with fidelity/cost estimates.
- **QMARL Training Service**: orchestrates centralized training and pushes policy artifacts to QCA agents and to PLM.
- **Quantum Simulation Sandbox**: high-throughput simulator farm (batched rollouts) for planners/QMARL trainers.

MIDDLEWARE

- Ensure low-latency paths for hybrid optimizer-quantum loops (favor gRPC + fast runtimes like Qiskit Runtime).

WORKER POOLS (Execution Plane) — new specialized workers

- **QCA Agents**: containers embedding quantum calls; used for quantum-assisted decision-making. Tag with `qca` capability.
- **QMARL Agents**: runtime hosts where deployed policy circuits run; can be on edge/classical workers or invoke QPU for evaluation.
- **AutoCircuit Workers**: run circuit generation/search/evaluation pipelines; produce ranked artifacts saved to PLM.
- **QPU Broker**: extended to support topology-aware transpilation & ansatz-aware compilation.

ADAPTERS (runtime descriptors)

- Must advertise:
  - native_topology, supports_dynamic_circuits, mid_circuit_measurements, parametric_circuits, runtime_latency_estimates, fidelity_profile.
- QPU Broker uses these to map HVA/QAOA ansatzes and choose transpilation strategies.

SIMULATION SANDBOX

- Provides batched rollout APIs for Planner/QMARL trainers to get expected utility, variance, or train policies without consuming costly QPU cycles.

INFRA / DEPLOYMENT

- Add CRD fields for `maturity` and `capabilities` so k8s / Helm can deploy staging vs production agent sets separately.

MONITORING / PLM / EVALUATION

- Add new dashboards & benchmarks:
  - QMARL learning curves (episodes → reward), sample complexity charts.
  - QCA decision latency / fidelity metrics.
  - AutoCircuit ranking, cost vs fidelity tradeoffs.
- PLM must store: ansatz template version, autotuner parameters, QMARL checkpoint ID, allowed runtimes per maturity stage.

Quick canonical job flow (updated)

1. Client selects job + maturity stage → API Gateway validates.
2. Hub Intake → Planner consults Simulation Sandbox (rollouts) and AutoCircuit Designer (if circuit needed) → produces a plan.
3. Splitter classifies steps (classical / quantum_subroutine / qca_task / simulation) and tags each with maturity & required adapter capabilities.
4. Scheduler enforces maturity policy and maps steps to:
   - Classical Workers,
   - Hybrid Executors,
   - QCA containers,
   - QMARL training queues,
   - QPU Broker (with adapters).
5. QPU Broker compiles/transpiles (topology-aware, ansatz-aware) or uses AutoCircuit-provided circuits; QCA agents may run internal quantum subroutines.
6. Results & provenance (agent IDs, policy circuit IDs, adapter transformations, noise/fidelity metadata) → Aggregator → Artifact Store & PLM.
7. Client receives final result + explainability & maturity-aware notes (e.g., "Prototype run on staging QPU; results noisy; recommended next steps").

Maturity Model (short)

- **Concept/Dev (simulator-only)** — local dev & sandbox.
- **Prototype (staging)** — limited QPU shots, human approval for high-cost runs.
- **Validated (production)** — constrained production QPU access, automated monitoring.
- **Certified (SLA)** — full production access with governance, audit, and hardened policies.

Planner & Scheduler use these stages to decide what resources/agents/runtimes are allowed.

//

What I created (all shown inline)

- Tables:
  - Subtasks summary (type, assigned agent, mean, shots, backend).
  - Pipeline logs (RAG, RPC, API, ID verification, Planner, QPU Broker, QCA decision, Aggregator).
  - Runtime KPIs (queue length, latency, QPU queue wait).
  - Internal products table (QAI Processor, QAI OS, QAI Datacenter — initialized with dummy values).
  - End-to-end trace (component, event, duration).
  - KPI summary (total subtasks, backends used, avg shots, artifact path).
- Charts:
  - Histogram of sample distribution (synthetic if raw samples not present).
  - Queue length over time.
  - Average latency over time.
  - Backend selection pie chart.
- Simulated workflow trace entries covering: Web service/API Gateway, ID verification, RAG retrieval, Crew/Planner steps, AutoCircuit proposals, Simulation sandbox rollouts, QPU selection/submission, QCA decision, Aggregator/PLM persistence.

Notes & next steps

- Visuals are generated from the artifact JSON at `/mnt/data/qai_demo_artifacts/job_89fba2cd.json`. If you run the Colab demo and produce a new artifact, re-run the visualization cell to update charts.
- I initialized internal products (QAI Processor, OS, Datacenter) with dummy metadata — when you provide real endpoints, we can replace these and show live health/status metrics.
- If you prefer different KPI metrics (cost, fidelity, error rate, job success), tell me which ones and I'll add them.

//

For more details email: vijaymohire@gmail.com

// End