service_contracts.md (IronClaw)

This document defines **service boundaries**, **HTTP endpoints**, and **event types/payloads** for the IronClaw microservices architecture.

Principles:

- **Events are append-only** (ledger is the source of truth).
- **Artifacts live in git worktrees** (Vault service owns creation/removal).
- **Workers are stateless** (they write artifacts + AAR, then exit).
- **CO orchestrates** (plans, dispatches, synthesizes).

0) Shared identifiers and conventions

IDs

- `garrison_id` : string (e.g., `"local"`, `"prod-us1"`)
- `theater_id` : string (e.g., `"demo"`, `"proj_auth"`)
- `run_id` : ULID/UUID string
- `order_id` : ULID/UUID string
- `unit_id` : string (e.g., `"assault_abc123"`, `"fireteam_research"`)
- `worktree_id` : string (often same as `order_id`)
- `commit_sha` : git SHA string

Status enums

- `run_status`: `OPEN | COMPLETE | FAILED | CANCELLED`
- `order_status`: `QUEUED | CLAIMED | RUNNING | BLOCKED | COMPLETED | FAILED | CANCELLED`
- `severity`: `LOW | MEDIUM | HIGH | CRITICAL`

Time

- Use ISO-8601 UTC: `2026-01-14T16:21:00Z`

Response envelopes (recommended)

All services may respond with:

```
{ "ok": true, "data": {...}, "error": null }
```

On error:

```
{ "ok": false, "data": null, "error": {"code": "...", "message": "..."} }
```

1) Event types (Ledger canonical)

Events are immutable. Consumers derive state from events + snapshots.

Required base event fields

```
{
  "event_id": "01J...",
  "ts": "2026-01-14T16:21:00Z",
  "type": "ORDER_CREATED",
  "garrison_id": "local",
  "theater_id": "demo",
  "run_id": "01J...",
  "order_id": "01J...",
  "unit_id": null,
  "payload": {}
}
```

Core event types

Run lifecycle

- `RUN_CREATED`
- `RUN_UPDATED` (objective/metadata change)
- `RUN_COMPLETED`
- `RUN_FAILED`
- `RUN_CANCELLED`

Order lifecycle

- `ORDER_CREATED`
- `ORDER_ENQUEUED`
- `ORDER_CLAIMED`
- `ORDER_STARTED`
- `ORDER_BLOCKED`
- `ORDER_COMPLETED`
- `ORDER_FAILED`
- `ORDER_REISSUED`
- `ORDER_CANCELLED`

Vault/worktree

- `WORKTREE_CREATED`
- `WORKTREE_READY`
- `WORKTREE_ARCHIVED`
- `WORKTREE_REMOVED`

Artifacts & reporting

- `ARTIFACT_WRITTEN`
- `AAR_WRITTEN`

Integration

- `INTEGRATION_READY`
- `INTEGRATION_STARTED`
- `INTEGRATION_PASSED`
- `INTEGRATION_FAILED`
- `INTEGRATED`

Escalation / recovery

- `ESCALATION_RAISED`
- `ESCALATION_ACKED`
- `RECOVERY_REQUIRED`
- `RECOVERY_STARTED`
- `RECOVERY_COMPLETED`

System health

- `PATROL_TICK` (DO/Observer heartbeat)
- `SERVICE_DEGRADED`
- `SERVICE_RECOVERED`

2) Artifact formats (shared)

Order file (in worktree)

Path: `order.json`

```
{
  "order_id": "order_001",
  "run_id": "run_123",
  "theater_id": "demo",
  "type": "draft",
  "objective": "Summarize the input into 5 bullets",
  "inputs": { "text": "..." },
  "constraints": {
    "budget_seconds": 60,
    "max_retries": 1,
    "tool_policy": { "network": false, "fs_allowlist": ["./"] }
  },
  "profile": { "name": "executor_default", "model": "meta-llama/..." }
}
```

AAR file (in worktree)

Path: `aar.json`

```
{
  "order_id": "order_001",
  "status": "completed",
  "completed_at": "2026-01-14T16:21:00Z",
  "summary": "What was done and results",
  "artifacts": [
    { "path": "outputs/model_output.txt", "type": "text/plain" }
  ],
  "model": {
    "provider": "io.intelligence",
    "name": "meta-llama/Llama-3.3-70B-Instruct",
    "temperature": 0.2
  },
  "notes": [],
  "next_recommendations": []
}
```

3) Services and contracts

3.1 api_gateway (Chat surface)

Purpose: user-facing entrypoint.

POST /chat Request:

```
{
  "user_id": "u_123",
  "theater_hint": "demo",
  "message": "Build me a summary",
  "mode": "sync"
}
```

Response (sync):

```
{
  "ok": true,
  "data": {
    "run_id": "01J...",
    "reply": "Final user-facing response text",
    "artifacts": []
  },
  "error": null
}
```

Response (async):

```
{
  "ok": true,
  "data": { "run_id": "01J...", "status": "OPEN" },
  "error": null
```

```
}
```

GET /runs/{run_id} Returns run status + summary.

GET /orders/{order_id} Returns order status + pointers to artifacts/AAR.

3.2 co_service (Commanding Officer)

Purpose: planning, routing, dispatch, synthesis.

POST /runs Create a run from user input (usually called by `api_gateway`). Request: `{ "user_id": "u_123", "theater_hint": "demo", "objective": "..." }`

POST /runs/{run_id}/plan Produces 1–N orders (METE). Response includes created `order_ids` and dependencies.

POST /runs/{run_id}/dispatch Enqueues orders to worker runtime / workflow engine.

POST /runs/{run_id}/synthesize Reads artifacts/AARs and produces final reply.

CO emits events to ledger_service: `RUN_CREATED`, `ORDER_CREATED`, `ORDER_ENQUEUED`, `RUN_COMPLETED`, etc.

3.3 ledger_service (Events + snapshots)

Purpose: single source of truth for state.

POST /events Append one event. Request:

```
{
  "type": "ORDER_STARTED",
  "garrison_id": "local",
  "theater_id": "demo",
  "run_id": "01J...",
  "order_id": "01J...",
  "unit_id": "assault_abc123",
  "payload": { "attempt": 1 }
}
```

POST /events/batch Append many events at once (recommended for workers).

GET /runs/{run_id} Returns run + derived snapshot.

GET /orders/{order_id} Returns order + derived snapshot + latest known artifact pointers.

GET /health Healthcheck.

Ledger guarantees:

- events are stored durably
- idempotency supported via `event_id` (recommended)

3.4 vault_service (Theater repo + worktrees)

Purpose: create/prepare/cleanup worktrees and write initial operational files.

POST /theaters Create/open a Theater repo. Request: `{ "theater_id": "demo", "repo_path": "/.../theaters/demo/repo" }`

POST /worktrees Create a worktree for an order. Request: `{ "theater_id": "demo", "order_id": "01J...", "branch": "order_01J..." }` Response:

```
{
  "ok": true,
  "data": {
    "worktree_path": "/.../theaters/demo/worktrees/order_01J...",
    "branch": "order_01J..."
  },
  "error": null
}
```

Emits events: `WORKTREE_CREATED`, `WORKTREE_READY`

POST /worktrees/{order_id}/prepare Write `order.json`, `task.md`, `state.json` (optional). Request: `{ "order": { ... } }`

POST /worktrees/{order_id}/archive Archives completed worktree (zip, move, tag) per policy.

DELETE /worktrees/{order_id} Removes worktree safely (git worktree remove).

3.5 worker_service (Assault Units / Fireteams runtime)

Purpose: execute orders, call `io.intelligence`, write artifacts + AAR. Workers consume jobs from the queue/workflow engine (not necessarily HTTP).

If HTTP-controlled, use: **POST /jobs** Request:

```
{
  "order_id": "01J...",
  "theater_id": "demo",
  "run_id": "01J...",
  "unit_id": "assault_abc123",
  "worktree_path": "/.../worktrees/order_01J...",
  "profile": { "name": "executor_default", "model": "meta-llama/..." },
  "tool_policy": { "network": false }
}
```

Worker actions (contract):

1. Load secrets (dotenv / injected)
2. Call `io.intelligence` (OpenAI-compatible)
3. Write artifacts to `outputs/`
4. Write `aar.json`
5. Commit changes
6. Emit events batch: `ORDER_STARTED`, `ARTIFACT_WRITTEN`, `AAR_WRITTEN`, `ORDER_COMPLETED` (or `ORDER_FAILED`)

3.6 observer_service (Theater oversight)

Purpose: detect stalls, validate completion, trigger recovery.

POST /theaters/{theater_id}/patrol Runs one patrol tick. Checks:

- Orders stuck in `RUNNING` too long
- Worktrees exist but no recent commits
- Missing AAR/artifacts on "completed"
- Dirty worktree states

Emits:

- `RECOVERY_REQUIRED` (with reason)
- `ESCALATION_RAISED` (severity, route)
- `INTEGRATION_READY` when AAR + artifacts valid

3.7 integration_service (Validation + promotion)

Purpose: apply gates and promote changes from worktree to Theater mainline.

POST /integrate Request:

```
{
  "theater_id": "demo",
  "order_id": "01J...",
  "worktree_path": "/.../worktrees/order_01J...",
  "gate_profile": "default"
}
```

Actions:

1. Validate `aar.json` schema
2. Ensure artifacts exist
3. Run configured gates (tests/lint/policy)
4. Promote via merge/cherry-pick

Emits: `INTEGRATION_STARTED`, `INTEGRATION_PASSED` or `INTEGRATION_FAILED`, `INTEGRATED` (with commit SHA)

3.8 do_service (Duty Officer)

Purpose: system-wide patrol and recovery.

POST /patrol One patrol tick:

1. Scan ledger for overdue orders
2. Reissue orders within retry caps
3. Request observer patrols
4. Request worktree quarantine if needed

Emits:

- `PATROL_TICK`
- `RECOVERY_STARTED`, `ORDER_REISSUED`, `RECOVERY_COMPLETED`

- `SERVICE_DEGRADED` if dependencies failing

3.9 sentinel_service (Watchdog auditor)

Purpose: confirm DO is alive and effective.

POST /audit

- Checks last `PATROL_TICK`
- Checks backlog trend
- Alerts or triggers failover if needed

Emits:

- `ESCALATION_RAISED` (CRITICAL) if DO is dead or ineffective

3.10 learning_service (Policy updates)

Purpose: outcome-based routing updates (auditable).

POST /learn/tick Consumes recent events and computes:

- Success rate by profile/task_type
- Latency distribution
- Retry counts
- User correction signals (if logged)

Outputs:

- Updated routing recommendations (policy doc)
- Emits `RUN_UPDATED` (policy version) or `POLICY_UPDATED` (optional event type)

4) Queue/Workflow payloads (recommended)

Job message (for worker queue)

```
{
  "job_type": "EXECUTE_ORDER",
  "order_id": "01J...",
  "run_id": "01J...",
  "theater_id": "demo",
  "unit_id": "assault_abc123",
  "worktree_path": "/.../worktrees/order_01J...",
  "attempt": 1,
  "profile": { "name": "executor_default", "model": "meta-llama/..." },
  "constraints": { "budget_seconds": 60, "max_retries": 1 },
  "tool_policy": { "network": false, "fs_allowlist": ["./"] }
}
```

Job result (emitted as events, not returned)

Use `POST /events/batch` with `ORDER_COMPLETED` or `ORDER_FAILED` plus artifact/AAR pointers.

5) Minimal MVP subset (if you want to cut scope)

To ship MVP with microservices, you can start with only:

1. `api_gateway` (or skip if using CLI)
2. `co_service`
3. `ledger_service`
4. `vault_service`
5. `worker_service`