AOAPI v0.1 (Agent-Oriented API)

A minimal protocol proposal and reference patterns for reliable agent execution

Monica King, J.D. — Coherence Protocol — January 04, 2026

Abstract — UI-driven agents fail in production because success is inferred from perception. AOAPI shifts execution from perception-based guessing to protocol-based atomic actions, each returning a verifiable receipt (state hash + optional UI proof + signature). This turns hallucination into verification.

1. Problem: Perception-Based Execution Does Not Converge

Current “computer use / browser automation” agents must parse unstable DOMs, async UI shifts, A/B tests, and anti-bot defenses. Reliability degrades as surface area grows; engineering effort scales linearly with web entropy. Enterprise buyers need *non-bypassable* pre-commit verification, not more UI scraping.

2. AOAPI Core Idea

Sites expose a machine-readable descriptor at /.well-known/aoapi.json. Agents discover supported atomic actions (e.g., search_patent) with input/output schemas. Each action returns a **receipt** containing verifiable proofs (state hash, optional UI digest, signature) so clients can validate state transitions.

2.1 Minimal Descriptor Example

```
{
  "aoapi_version": "0.1",
  "service": {
    "name": "Example Patent Portal",
    "base_url": "https://example.com"
  },
  "discovery": {
    "well_known": "/.well-known/aoapi.json"
  },
  "actions": [
    {
      "name": "search_patent",
      "description": "Search patents by keywords/assignee/inventor and return normalized results.",
      "method": "POST",
      "path": "/aoapi/actions/search_patent",
      "input_schema": {
        "type": "object",
        "properties": {
          "query": {
            "type": "string"
          },
          "assignee": {
            "type": "string"
          },
          "inventor": {
            "type": "string"
          },
          "limit": {
            "type": "integer",
            "default": 10,
            "minimum": 1,
```

```
            "maximum": 50
          }
        },
        "required": [
          "query"
        ]
      },
      "output_schema": {
        "type": "object",
        "properties": {
          "results": {
            "type": "array"
          },
          "receipt": {
            "type": "object"
          }
        },
        "required": [
          "results",
          "receipt"
        ]
      },
      "verification": {
        "mode": "shadow_ui",
        "ui_proof": {
          "type": "dom_digest",
          "selector": "#results",
          "hash_alg": "sha256"
        },
        "receipt_sig": {
          "alg": "ed25519",
          "kid": "portal-key-2026-01"
        }
      }
    }
  ]
}
```

3. Receipt: Execution Proof as a First-Class Primitive

An AOAPI action MUST return a receipt. Minimum fields: state_hash, timestamp, nonce. Optional: ui_hash (DOM/screenshot digest), signature. Clients treat “success” as untrusted unless receipt verification passes. This enables replay protection, auditability, and deterministic recovery.

4. OpenAI Integration Pattern: Schema-Driven Tool Routing

OpenAI’s tool calling excels when the environment is deterministic. AOAPI makes the environment deterministic by exposing action schemas and receipts. The model emits an action plan; the orchestrator executes AOAPI calls; the model consumes tool results.

```
// Pseudocode (Node.js): model selects AOAPI action, orchestrator executes, then feeds result back.
const tools = [{
  type: "function",
  name: "aoapi_call",
  description: "Call an AOAPI atomic action on a target site and return results + verifiable receipt.",
  parameters: {
    type: "object",
    properties: {
      base_url: { type: "string" },
      action: { type: "string" },
```

```
      input:  { type: "object" }
    },
    required: ["base_url","action","input"]
  }
}];

// Orchestrator executes:
// POST `${base_url}/aoapi/actions/${action}` with `input`
// then verify receipt.state_hash/ui_hash/signature before returning tool_result.
```

5. Anthropic Integration Pattern: Computer Use as a Verified Fallback

Claude’s computer use is perception-first. AOAPI inverts the burden: the interface self-describes actions and returns proofs. When AOAPI is present, Claude can “look once” for confirmation; otherwise, it falls back to computer use while still producing receipts by hashing screen states for audit.

```
def run_task(task, base_url):
    aoapi = try_fetch(f"{base_url}/.well-known/aoapi.json")
    if aoapi and has_action(aoapi, "search_patent"):
        out = http_post(f"{base_url}/aoapi/actions/search_patent", {"query": task, "limit": 5})
        verify_receipt(out["receipt"])   # signature + nonce + timestamp window
        return out
    return computer_use_fallback(task)   # still produce UI-hash receipts per step
```

Appendix A — Demo Scenario: Patent Search (Vertical-First)

Why patents: high-value workflows, complex search forms, brittle portals, strong audit needs. AOAPI turns a messy UI into a deterministic action with a signed receipt suitable for legal/compliance-grade logs.

```
# FastAPI reference: /aoapi/actions/search_patent returns results + signed receipt
from fastapi import FastAPI
from pydantic import BaseModel, Field
import hashlib, json, time, os
from nacl.signing import SigningKey

app = FastAPI()
SIGNING_KEY = SigningKey(bytes.fromhex(os.environ["AOAPI_ED25519_SK_HEX"]))

class SearchPatentIn(BaseModel):
    query: str
    assignee: str | None = None
    inventor: str | None = None
    limit: int = Field(default=10, ge=1, le=50)

def sha256_hex(b: bytes) -> str:
    return hashlib.sha256(b).hexdigest()

@app.post("/aoapi/actions/search_patent")
def search_patent(payload: SearchPatentIn):
    results = [{"publication_number":"US-2026-0001234",
                "title":f"Result for: {payload.query}",
                "assignee":payload.assignee or "",
                "filing_date":"2026-01-01"}][: payload.limit]
    state_obj = {"action":"search_patent","payload":payload.model_dump(),"results":results}
    state_hash = sha256_hex(json.dumps(state_obj, sort_keys=True).encode())
    ui_hash = "sha256:TODO_dom_digest"
    nonce = sha256_hex(os.urandom(16))
    ts = time.strftime("%Y-%m-%dT%H:%M:%SZ", time.gmtime())
    receipt = {"state_hash":state_hash,"ui_hash":ui_hash,"timestamp":ts,"nonce":nonce}
    sig = SIGNING_KEY.sign(json.dumps(receipt, sort_keys=True).encode()).signature.hex()
    receipt["signature"] = sig
    return {"results": results, "receipt": receipt}
```

Appendix B — Receipt Verification (Shadow Verifier)

```
def verify_receipt(receipt, pubkey):
    # 1) signature verification (ed25519) over canonical JSON
    # 2) timestamp window check (e.g., +/- 60s)
    # 3) nonce replay protection (nonce store)
    # 4) optional UI hash cross-check if shadow_ui enabled
    return True
```