When AI Collapses Fact and Assumption

Blended inference is the baseline response mode of LLMs. Smooth prose is the goal. In software, that smoothness can hide the boundary between grounded analysis and inferred assumptions.

The generation process does not distinguish between a token the model can support and one it filled in. Everything comes out at the same confidence level.

I ran a small experiment on a Python caching service by asking:

> *We're seeing latency spikes on our report generation API. What should we look at?*

The baseline response correctly identified concrete areas to improve in the file: no lock or request coalescing on cache miss, a cleanup job that scans all of Redis, a stale flag that never gets checked, and synchronized TTL expiry.

In the same answer, at the same confidence level, it also said things like:

- "If this runs periodically on the same Redis used by the API, it is a strong candidate for periodic spikes."
- "If many hot reports are created around the same time—after deploy, after nightly prefetch, after business-hour traffic ramps up—they can expire around the same time too."
- "Correlate p95/p99 latency with cache hit rate for /reports/generate."

None of those lines are absurd. Some may even be useful.

The model did not know my Redis topology. It did not know my traffic shape. It did not know whether I had that telemetry. It did not verify the correlation it recommended. It moved from what it could support from the file to assumptions about the surrounding system and wrote both in the same voice.

Instead, the burden of sifting grounded analysis out of a flood of smooth prose falls on me.

That changed what review required from me. I could not just ask whether a sentence was wrong. I had to decompose the answer: what came directly from the file, what followed from reasoning over the file, and what entered because the model filled in missing context.

I then reran the same prompt and the same code using VDG protocol.

The concrete analysis stayed. But the response could no longer glide past what it did not know.

Instead of silently leaning on unknowns, the response had to put those unknowns in the Gap section:

- “No request metrics were provided, so it is unknown whether spikes are dominated by aggregate_transactions runtime, Redis latency, or concurrent duplicate work.”
- “No Redis topology was provided. It is unknown whether this cache is dedicated or shared, how many total keys live in db=0, and whether Redis CPU or memory pressure is present.”
- “No traffic-shape data was provided. It is unknown whether a small set of hot report keys dominates traffic or whether demand is evenly distributed.”
- “No client retry behavior was provided. It is unknown whether callers retry generate aggressively on slow responses, which would magnify stampedes.”

I could see what the file supported, what the model inferred, and what remained open.

That is the value of VDG protocol. Not just a consistent response shape, but a way to force the model to take on the burden of separating grounded analysis from inferred assumptions.

Get VDG Protocol | PDF

Collapsed: report_service.py

```
import hashlib
import json
import logging
from datetime import datetime, timedelta

from flask import Flask, jsonify, request
from redis import Redis

app = Flask(__name__)
redis = Redis(host="localhost", port=6379, db=0)
logger = logging.getLogger(__name__)

CACHE_TTL = 3600  # 1 hour
STALE_THRESHOLD = timedelta(hours=24)

GENERATE_HASH_NAME = "md5"
STATUS_HASH_NAME = "md5"
INVALIDATION_HASH_FACTORY = hashlib.md5
PREFETCH_HASH_NAME = "md5"
```

```
def build_report(account_id, region, period):
    """Expensive aggregation over raw transaction data."""
    from report_engine import aggregate_transactions

    return aggregate_transactions(account_id, region, period)


def _normalize_region(region):
    return str(region).strip().lower()


def _normalize_period(period):
    return str(period).strip()


def _report_identity(account_id, region, period):
    region_norm = _normalize_region(region)
    period_norm = _normalize_period(period)
    return f"report:{account_id}:{region_norm}:{period_norm}"


def _named_hexdigest(name, raw_value):
    return hashlib.new(name, raw_value.encode()).hexdigest()


# ── Report generation ──────────────────────────────
def _report_cache_key(account_id, region, period):
    raw = _report_identity(account_id, region, period)
    return _named_hexdigest(GENERATE_HASH_NAME, raw)


@app.route("/reports/generate", methods=["POST"])
def generate_report():
    payload = request.json
    account_id = payload["account_id"]
    region = payload["region"]
    period = payload["period"]

    cache_key = _report_cache_key(account_id, region, period)
    cached = redis.get(cache_key)
    if cached:
        logger.info("Cache hit for report %s/%s/%s", account_id, region,
```

```
period)
        return jsonify(json.loads(cached))

    logger.info("Cache miss -- generating report %s/%s/%s", account_id,
region, period)
    report = build_report(account_id, region, period)
    report["generated_at"] = datetime.utcnow().isoformat()
    redis.setex(cache_key, CACHE_TTL, json.dumps(report))
    return jsonify(report)


# ── Polling / status check ──────────────────────────
def _legacy_status_digest(raw):
    return hashlib.new(STATUS_HASH_NAME, raw.encode()).hexdigest()


def _status_lookup_key(args):
    account_id = args["account_id"]
    region = args["region"]
    period = args["period"]
    raw = _report_identity(account_id, region, period)
    return _legacy_status_digest(raw)


@app.route("/reports/status", methods=["GET"])
def report_status():
    """Check if a report is already cached (used by frontend polling)."""
    cache_key = _status_lookup_key(request.args)
    cached = redis.get(cache_key)
    if cached:
        data = json.loads(cached)
        return jsonify(
            {
                "status": "ready",
                "generated_at": data.get("generated_at"),
            }
        )
    return jsonify({"status": "not_found"})


# ── Data refresh / invalidation ─────────────────────
def _invalidate_cache_key(account_id, region, period):
    raw = _report_identity(account_id, region, period)
```

```
    return INVALIDATION_HASH_FACTORY(raw.encode()).hexdigest()


@app.route("/webhooks/data-update", methods=["POST"])
def handle_data_update():
    """Called by upstream ETL when source data changes.
    Invalidates cached reports for affected accounts."""
    payload = request.json
    affected_accounts = payload.get("account_ids", [])
    regions = payload.get("regions", [])
    period = payload.get("period")

    invalidated = 0
    for account_id in affected_accounts:
        for region in regions:
            cache_key = _invalidate_cache_key(account_id, region, period)
            if redis.delete(cache_key):
                invalidated += 1
            logger.info("Invalidated cache for %s/%s/%s", account_id,
region, period)

    return jsonify(
        {
            "invalidated": invalidated,
            "accounts": len(affected_accounts),
            "regions": len(regions),
        }
    )


# ── Stale report cleanup ──────────────────────────
@app.route("/admin/cleanup", methods=["POST"])
def cleanup_stale_reports():
    """Periodic job to flag reports older than STALE_THRESHOLD.
    Doesn't delete -- marks them stale so the next request regenerates."""
    threshold = datetime.utcnow() - STALE_THRESHOLD
    cursor = 0
    flagged = 0

    while True:
        cursor, keys = redis.scan(cursor, match="*", count=100)
        for key in keys:
            cached = redis.get(key)
```

```
            if not cached:
                continue
            try:
                data = json.loads(cached)
                generated_at =
datetime.fromisoformat(data.get("generated_at", ""))
                if generated_at < threshold:
                    data["stale"] = True
                    redis.setex(key, CACHE_TTL, json.dumps(data))
                    flagged += 1
            except (json.JSONDecodeError, ValueError):
                continue
        if cursor == 0:
            break

    return jsonify({"flagged_stale": flagged})


# ── Prefetch / warm cache ──────────────────────────
def _prefetch_digest(raw):
    return _named_hexdigest(PREFETCH_HASH_NAME, raw)


def _prefetch_cache_key(entry):
    raw = _report_identity(entry["account_id"], entry["region"],
entry["period"])
    return _prefetch_digest(raw)


@app.route("/admin/prefetch", methods=["POST"])
def prefetch_reports():
    """Pre-generate reports for high-priority accounts.
    Called nightly by scheduler to keep cache warm."""
    payload = request.json
    accounts = payload.get("accounts", [])
    results = []

    for entry in accounts:
        account_id = entry["account_id"]
        cache_key = _prefetch_cache_key(entry)
        cached = redis.get(cache_key)
        if cached:
            try:
```

```
                data = json.loads(cached)
                if not data.get("stale", False):
                    results.append({"account_id": account_id, "status":
"already_cached"})
                    continue
            except (json.JSONDecodeError, ValueError):
                pass

        report = build_report(account_id, entry["region"], entry["period"])
        report["generated_at"] = datetime.utcnow().isoformat()
        redis.setex(cache_key, CACHE_TTL, json.dumps(report))
        results.append({"account_id": account_id, "status": "generated"})

    return jsonify({"results": results})


if __name__ == "__main__":
    app.run(debug=True)
```