**Question 1.2**

**1. Performance**

1. **Asynchronous I/O**
   * FastAPI + Uvicorn for non‑blocking request handling.
   * SQLAlchemy’s AsyncSession to avoid thread blocking on DB calls.
2. **Caching**
   * Redis used to cache list endpoint results (TTL=60 s).
   * Cache key patterns include query parameters to serve identical requests from cache.
3. **Pagination & Filtering**
   * Cursor‑based pagination (limit + cursor) to avoid large scans.
   * DB-side LIMIT and indexed filters ensure efficient row retrieval.
4. **Connection Pooling & Read Scaling**
   * Persistent connection pool in SQLAlchemy engine.
   * Read replicas for scaling heavy read workloads.
5. **Compression & HTTP/2**
   * GZip middleware for payloads >1 KB.
   * Provision for HTTP/2 to reduce request overhead.

**2. Reliability**

1. **Rate Limiting & Throttling**
   * fastapi-limiter with Redis enforces per‑endpoint quotas (e.g., 10 req/min for list, 20 req/min for detail).
   * Guards against traffic spikes and abuse.
2. **Schema Migrations**
   * Alembic migrations run at startup or CI/CD, ensuring DB schema consistency.
   * create\_tables.py as a bootstrap fallback.
3. **Circuit Breakers & Retries**
   * Outbound HTTP calls (if any) wrapped in retry logic with exponential backoff.
   * DB transactions retried on transient failures (e.g., deadlocks).
4. **Horizontal Scalability**
   * Stateless API containers behind a Load Balancer enables seamless scaling.
   * Kafka consumer groups for the processor allow parallel, balanced message handling.
5. **Observability & Alerting**
   * Structured logging (JSON) with request‑ID, latency, status.
   * Metrics (Prometheus + Grafana): latency, error rates, DB pool usage, cache hit ratio, Kafka lag.
   * Alerts on consumer lag or error‑rate spikes.

**3. Security**

1. **Transport Security**
   * TLS/TCP for all ingress and internal service communication.
   * HSTS and secure cookie flags if sessions are used.
2. **Input Validation & Injection Protection**
   * Pydantic schemas validate all request payloads and query parameters.
   * SQLAlchemy’s parameterized queries to prevent SQL injections.
3. **Authentication & Authorization**
   * OAuth 2.0 / JWT tokens for authenticated access (extensible via FastAPI dependencies).
   * Role‑based access control enforced per endpoint.
4. **Rate‑Limiting & Abuse Prevention**
   * Dual per‑IP and per‑API‑key limits.
   * Auto‑ban or CAPTCHA challenge on repeated violations.
5. **Secrets Management**
   * All credentials (DB, Kafka, Redis) are to be stored in environment variables or a secret vault.
   * No hard‑coded secrets in source code or container images.
6. **Dependency & Image Hardening**
   * Regular dependency audits (Dependabot, pip-audit).
   * Minimal base images (e.g., Alpine Linux) and non‑root containers.
7. **Security Headers & CORS**
   * Strict CORS policy (whitelist known origins).
   * Headers: Content-Security-Policy, X-Content-Type-Options, X-Frame-Options.

**Question 1.3**

**Implementing Parallel & Multi‑Threaded Processing for Large Data Volumes**

1. **Topic Partitioning & Consumer Scaling**

* Kafka partitions: by splitting each topic (e.g. raw‑pokemon) into P partitions using a hash of the entity ID.
* Consumer group: by running C processor instances in the same group, Kafka will automatically distribute partitions.
* Scale‑out: adding more consumers gives near‑linear throughput gains.

**2. Parallel Ingestion (I/O‑Bound)**

* Async HTTP fetch: use asyncio + httpx.AsyncClient with a Semaphore(N) to limit concurrent requests:

sem = asyncio.Semaphore(50) # max 50 in‑flight HTTP calls

async def fetch(url):

async with sem, httpx.AsyncClient() as client:

return await client.get(url)

responses = await asyncio.gather(\*(fetch(u) for u in urls))

* Batch Kafka produce: group ~100 records and await producer.send\_and\_wait() in parallel to amortize network and broker I/O.

**3. Concurrent Transformation (Mixed I/O & CPU)**

* Async task pool: inside the consumer loop, dispatch each message to an asyncio.create\_task, gated by a Semaphore(M):

sem = asyncio.Semaphore(100)

async for msg in consumer:

await sem.acquire()

asyncio.create\_task(handle(msg, sem))

* CPU‑Bound offload: wrap heavy computations in a ProcessPoolExecutor:

executor = ProcessPoolExecutor(max\_workers=4)

result = await loop.run\_in\_executor(executor, cpu\_heavy\_fn, data)

* ThreadPool for light CPU: for moderate CPU usage (e.g. JSON parsing), a ThreadPoolExecutor can help without GIL contention.

**4. Parallel Storage Writes**

* Connection pooling: configure SQLAlchemy’s AsyncEngine with a pool size matching the concurrency level.
* Batch inserts collect transformed records and use INSERT … ON CONFLICT or multi‑row inserts to reduce per‑row overhead:

INSERT INTO pokemon (id,name,…) VALUES

(1,'Bulbasaur',…), (2,'Ivysaur',…)

ON CONFLICT (id) DO UPDATE SET name=EXCLUDED.name, …;

* Bulk ORM operations: if using SQLAlchemy Core, we can use await conn.execute(table.insert(), list\_of\_dicts).

**5. Autoscaling & Back‑Pressure**

* Kubernetes HPA: scale pods based on CPU or custom Kafka lag metrics.
* Internal semaphores: prevents over‑loading downstream (DB, network). If the semaphore is saturated, the consumer loop naturally slows.

**6. Monitoring & Tuning**

* By tracking throughput (msgs/sec), latency, error rates, DB pool usage and Kafka lag.
* By adjusting partition count, consumer count, semaphore limits, and executor pool sizes based on real‑world metrics.

**Question 1.4**

**Error Handling**

**1. Ingestion Layer**

* **Retrying with backoff**
  + Wrap all external HTTP/GQL calls in a retry policy (e.g. tenacity) with exponential backoff and jitter.
  + After N attempts, escalate: send the failed payload to a “dead‑letter” Kafka topic for later inspection.
* **Kafka produce errors**
  + Catch KafkaError on send\_and\_wait(), log the full exception along with message key/payload.
  + Retry a configurable number of times; on permanent failure, publish to a DLQ topic.

**2. Processor Layer**

* **Message deserialization**
  + Surround json.loads(msg.value) with try/except.
  + On JSON parse error, log the raw bytes at ERROR level, commit the offset (to avoid infinite retry), and move on.
* **Transformation & DB writes**
  + Wrap each database INSERT/UPSERT in a transaction.
  + Catch SQLAlchemyError (e.g. connection issues, constraint violations), roll back, log:
  + try:
  + await conn.execute(upsert\_stmt)
  + except SQLAlchemyError as e:
  + logger.error("DB write failed", error=str(e), record=data, partition=msg.partition)
  + # optional: push to DLQ or metrics counter
* **Graceful shutdown**
  + On unhandled exceptions in the main loop, stop the consumer cleanly to avoid offset loss, then exit.

**3. API Layer**

* **Input validation**
  + Leverage Pydantic models—invalid schemas automatically yield 422 responses.
* **Global exception handling**
  + FastAPI’s @app.exception\_handler(Exception) for catching unanticipated errors:
  + @app.exception\_handler(Exception)
  + async def internal\_error(request, exc):
  + logger.exception("Unhandled exception")
  + return JSONResponse(500, {"detail": "Internal server error"})
* **HTTPException usage**
  + Raise HTTPException(status\_code=404, detail="Not found") or 400 for client errors; these surface cleanly to the client without stack traces.
* **Timeouts & circuit breakers**
  + Enforce per‑route timeouts (e.g. via middleware) and wrap downstream calls in a circuit‑breaker (e.g., pybreaker) to fail fast if a dependency is unhealthy.

**Logging**

**Structured, Contextual Logging**

* JSON format for all logs, with fields:
  + timestamp (ISO 8601 UTC)
  + service (e.g. “api”, “processor”)
  + component (e.g. “db.write”, “kafka.consume”)
  + level (DEBUG/INFO/WARN/ERROR)
  + message
  + request\_id or correlation\_id
  + contextual data (query params, topic, partition, offset, payload IDs)

**Correlation IDs & Traceability**

* By generating a X-Request-ID at the API gateway; inject into all downstream logs and Kafka message headers.
* Propagate through ingestors, processor, and API so you can trace a single record end‑to‑end.

**Log Levels & Sampling**

* DEBUG for developer troubleshooting (startup, configuration values).
* INFO for normal operations (startup success, healthy heartbeats).
* WARN for recoverable anomalies (retrying an HTTP call).
* ERROR for failed operations (DB constraint violation, uncaught exceptions).
* Sampling: for very high‑volume logs (e.g. per‑message INFO logs), sample at a configurable rate to avoid log overload.

**Monitoring & Alerting**

1. **Metrics**
   * Counter for each error class (HTTP 5xx, JSON parses, DB errors, DLQ size).
   * Kafka consumer lag gauges.
   * Request/processing latencies (histograms).
2. **Health checks**
   * /health endpoint verifying DB connectivity and Redis availability.
   * Kubernetes readiness and liveness probes.
3. **Alerts**
   * High ERROR rate over 5 min window → PagerDuty.
   * Consumer lag > threshold → Scale up or investigate backlog.
   * Circuit‑breaker open state → Notify on degraded dependencies.