AutoDeck: Text-to-Anki Flashcard Service Report

1 Software and Hardware Components

The service is lightweight so that every tier can run on very basic Google Compute Engine VMs while keeping dependencies in the Python standard library plus a minimal set of supporting tools.

Component	Role	Notes / Configuration
Ubuntu 22.04 LTS (image family `ubuntu-2204-lts`)	Base OS image for all tiers	Provisioned once, snapshotted, and cloned for REST + worker VMs.
Python 3.10 + standard library	Runtime for Flask API, worker, and NLP pipeline	Pure-Python FK, TF-IDF, and regex-based NER implementations avoid heavy ML frameworks.
Flask 3.0 + Werkzeug	REST API tier	Handles upload/status/download endpoints. Deployed on a single VM tagged `allow-5000`.
Redis 5.x	Queue + metadata store	LPUSH/BRPOP job queue plus job status hashes/key-value pairs. Initially local to REST VM. designed to migrate to Cloud Memorystore.
NLP worker scripts (`worker.py`, `nlp.py`)	Transform text to decks	Workers consume queue jobs, run FK + lightweight pipeline, and emit CSV output.
Deployment tooling (`setup_base_vm.sh`, `create_rest_tier.py`, `create_workers.py`)	Provisioning + automation	Creates snapshots, firewall rules, and per-tier VMs with systemd-based startup scripts for robust process management.
Verification scripts (`verify_setup.sh`, `check_queue_process.sh`, `debug_rest_server.sh`)	Diagnostics and smoke tests	Validate gcloud config, VM state, systemd service status, environment variables, and REST endpoints after deployment.
REST VM hardware	`e2-medium` (2 vCPU, 4 GB RAM)	Hosts Flask + Redis + shared disk mounted at `/mnt/shared`.
Worker VMs	Two `f1-micro` instances (0.2 vCPU, 614 MB RAM)	Each pulls from Redis, reads file content from job metadata, processes text in 2–5 seconds, and writes CSV outputs locally.

Storage	Redis-based content passing + local disk	File content stored in Redis job metadata (up to 16MB). workers process from memory. Outputs written to local `/mnt/shared/outputs`. Upgrade path: Cloud Storage with signed URLs.
Networking	Firewall rule `allow-5000`, internal SSH	Public TCP/5000 for REST. Internal traffic between tiers over default VPC.

2 System Architecture and Interactions

Architectural Diagram

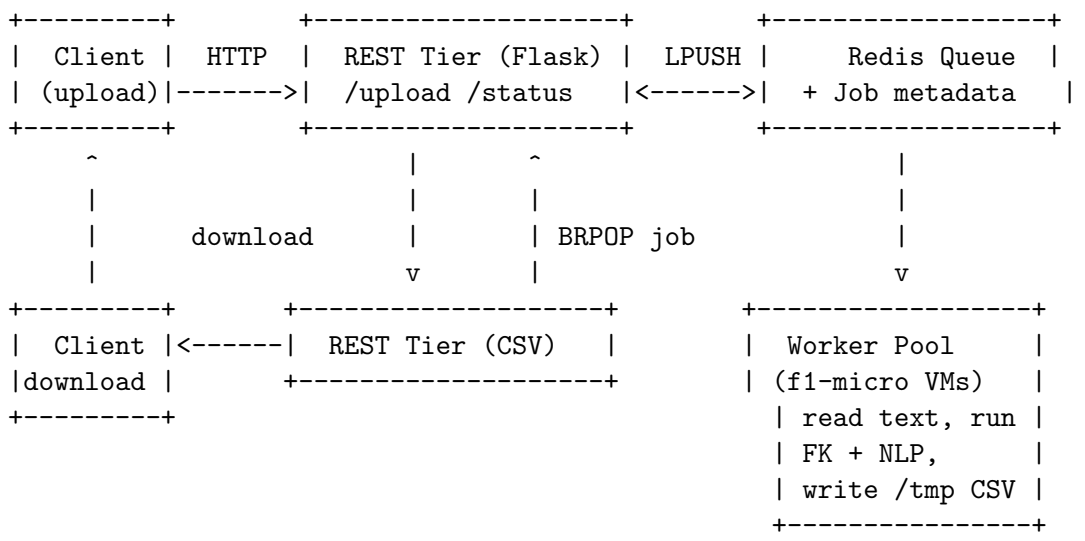

Figure 1: Logical architecture of the Text-to-Anki Service.

Interaction Narrative

1. Clients upload `.txt` files to the Flask REST tier, which saves them to local disk (`/mnt/shared/uploads`), reads the content into memory, stores both filepath and file content in Redis job metadata, and enqueues job IDs using `LPUSH`.
2. Workers running on separate `f1-micro` VMs perform blocking pops (`BRPOP`) against the `job_queue`, fetch the job metadata from Redis, and extract the `file_content` field directly from the metadata (no filesystem or network file transfer needed).
3. Each worker executes `nlp.py`. This calculates Flesch-Kincaid metrics, extracts keywords/entities, and emits complex-word CSV decks to its local. `/mnt/shared/outputs` directory.

4. Job status documents inside Redis are updated to reflect `queued → processing → completed/failed`, with output paths stored in metadata, enabling the REST tier to serve `/status/<job_id>` queries.
5. Clients download completed decks through `/download/<job_id>`, which streams the CSV from the REST server's local storage (workers' output files remain on worker VMs. This approach has limited capacity, a good future upgrade would be to sync to Cloud Storage like S3).

3 NLP Processing: Flesch-Kincaid Analysis

The core NLP processing pipeline uses Flesch-Kincaid Grade Level analysis to identify and extract the most complex vocabulary words from input texts.

Flesch-Kincaid Grade Level Formula

The Flesch-Kincaid Grade Level provides a readability score that corresponds to U.S. grade reading level:

$$FKGrade = 0.39 \times \left(\frac{totalwords}{totalsentences}\right) + 11.8 \times \left(\frac{totalsyllables}{totalwords}\right) - 15.59$$

This metric combines:

- **Average sentence length**: Longer sentences increase complexity
- **Average syllables per word**: Polysyllabic words indicate higher difficulty

Complex Word Extraction

Beyond calculating overall text grade level, this implementation performs word-level complexity analysis:

1. **Syllable counting**: Pure-Python heuristic counts vowel clusters in each word (e.g., "ephemeral" = 4 syllables, "cat" = 1 syllable)
2. **Complexity scoring**: Each unique word receives a complexity score: $complexity = syllables \times word\_length$
3. **Ranking and filtering**: Words are sorted by complexity score. the top N most difficult words (default: 20) are selected for flashcard generation
4. **Output format**: Complex words are written to CSV with placeholder definitions for user completion

Example complexity scores:

- *cat* (1 syllable × 3 letters) = 3 → excluded

- *dog* (1 syllable × 3 letters) = 3 → excluded
- *serendipity* (5 syllables × 11 letters) = 55 → included
- *ephemeral* (4 syllables × 9 letters) = 36 → included
- *obfuscate* (3 syllables × 9 letters) = 27 → included

Design Rationale

The pipeline is deliberately lightweight so it can run on `f1-micro` instances without external ML frameworks or pre-trained models, relying solely on pure-Python regex and arithmetic routines that finish within 2–5 seconds per document. The emphasis on FK-derived complexity scores keeps the generated flashcards focused on the terms most likely to challenge learners. The FK analysis replaced an earlier TF-IDF keyword keyword extractor and named-entity-recognition. It required some costly libraries like nltk, tensorflow, and keras that the worker VMs struggled to support. I found it added only marginal benefit compared to FK scores.

4 Debugging, Training, and Testing Process

Debugging workflow.

- The setup script now performs an SSH readiness loop before copying dependency installers.
- `verify_setup.sh` and `check_queue_process.sh` check that required gcloud services, firewall rules, VMs, and systemd services exist and are running before declaring successful deployment.
- `check_queue_process.sh` provides diagnostics: metadata server values, environment variables from running processes, shared storage access, and recent logs.
- Redis metadata allows replaying failed jobs by inspecting the stored file path and rerunning `worker.py` locally with the same payload.

Training/testing mechanisms.

- The NLP stack is deterministic and does not require model training. It's output can be verified with manual CLI runs ('python nlp.py sample-data/...').
- The README and deployment docs describe some simple tests. Running the pipeline against a fixed snippet, generating CSV output, and validating counts and keywords.
- Workers and the REST tier are tested with `quick-verify.sh` to ensure Redis connectivity, queue operations, and HTTP endpoints behave as expected after each deployment.

5 System Behavior, Capacity, and Bottlenecks

Working system explanation. A single `e2-medium` VM hosts the REST tier and Redis, exposing upload, status, and download endpoints while running under `anki-rest.service`. Two `f1-micro` workers join the queue via `BRPOP`. Configuration state such as `REDIS_HOST`, `SHARED_STORAGE_ROOT`,

SHARED_OUTPUT_FOLDER, and PYTHONUNBUFFERED is delivered through the service files while remaining accessible through the instance metadata. When a client uploads a file, the REST tier stores the text directly in Redis metadata (respecting the 16 MB limit) rather than relying on shared storage. Workers retrieve the file_content, execute FK in roughly 2–5 seconds, emit CSV decks into /mnt/shared/outputs, and updates the job status.

Supported workload.

- With 2 workers at 2–5 seconds per job, throughput is roughly 20–30 text documents per minute assuming small text files (<50 KB).
- The REST tier can handle concurrent uploads/status requests. CPU utilization remains low because heavy work is offloaded to workers.
- Scaling simply means cloning additional worker VMs from the same snapshot and pointing them at the shared Redis queue.

Potential bottlenecks. Having Redis and the filesystem on the REST VM is simple but also presents a single point of failure. Migrating to dCloud Storage would better ensure availability and make horizontal scaling feasible. Storing file content inside Redis further constrains scale because, although individual values can reach hundreds of megabytes, sustained ingestion of large files exhausts memory quickly, so the current 16 MB upload limit is a real contraint. The f1-micro workers also impose a constraint with 614 MB of RAM, unusually large texts could induce long runtimes, so input caps are necessary. Additionally, worker-generated CSVs remain on their local disks, meaning the REST tier must either share storage or synchronize outputs before clients can download results, another reason for eventual Cloud Storage adoption.

6 Summary

The Autodeck service prioritizes predictable deployment and datacenter-pattern reinforcement (snapshots, queues, systemd process management, monitoring scripts, shared storage mounts) over sophisticated NLP. By running everything on small Google Compute Engine VMs with pure Python tooling, the stack remains easy to debug, test, and reason about. Scaling paths (Cloud Memorystore, Cloud Storage, load-balanced REST tiers) are notable future improvements for this project.