ChatGPT

Technische Architectuur Blueprint

Deze blueprint beschrijft in detail hoe de Mustikarasa-systemen technisch opgezet en beheerd moeten worden. Het doel is een uitvoerbaar plan dat agnostisch is van specifieke bouwers of leveranciers, zodat elke implementator hiermee aan de slag kan. We behandelen alle aspecten – van tests en deployment tot configuratie, monitoring, security en upgrades – en werken deze tot in de puntjes uit, gebaseerd op de technische architectuur en principes van het systeem.

Systeemoverzicht

Het Mustikarasa-systeem is ontworpen als een **georkestreerde pipeline van AI-agents** binnen een streng governance-raamwerk. Een centrale **Orchestrator** coördineert een verzameling gespecialiseerde *worker*-agents die elk een specifieke taak uitvoeren op de content (recepten) [1] [2]. De workflow is vergelijkbaar met een assemblagelijn: elk AI-agent (*bijv.* vertaalcontrole, leesbaarheid, cultureel-historische annotatie, etc.) bewerkt of controleert het recept, waarna het resultaat naar de volgende agent gaat. Cruciaal is dat **traceerbaarheid** en **controle** ingebakken zijn in het ontwerp – elke wijziging wordt gelogd met reden, en gevoelige beslissingen vereisen menselijke goedkeuring (human-in-the-loop governance) [3] [4].

Technologie-stack: De implementatie is in **Python** met een op maat gemaakte framework-layer (bijv. `mustikarasa_agents.py` en gerelateerde scripts) om agents te beheren en de workflow te draaien [5]. De AI-agents worden aangedreven door **Large Language Models (LLM's)**. De architectuur is daarbij model-agnostisch: het systeem kan met verschillende LLM-backends werken – van OpenAI GPT-API's tot lokale modellen (zoals een Mistral-7B model via de **Ollama** runtime op een Mac) [6]. Deze modulariteit betekent dat men kan wisselen tussen een kleinere model (sneller/goedkoper) en een krachtiger model indien nodig, zonder de agent-logica zelf aan te passen.

Infrastructuur: Het systeem kan draaien op een enkele machine (bijvoorbeeld een **Mac Mini**) of in de cloud op een VM, zolang de benodigde omgeving beschikbaar is [7]. In onze opzet fungeert de Mac Mini als server waarop alle componenten draaien. De Orchestrator en agents draaien lokaal als processen of threads binnen de Python applicatie. Iedere agent neemt input (tekst, JSON) en stuurt een prompt naar het LLM, ontvangt een resultaat en schrijft dit weg. Tussen agents worden gegevens doorgegeven hetzij in-memory, hetzij via tijdelijke bestanden op schijf – bijvoorbeeld kan de output van agent A als JSON of Markdown worden opgeslagen, welke agent B vervolgens inleest [8]. De keuze voor bestanden vergemakkelijkt traceerbaarheid, maar voor performance kan waar nodig ook direct in-memory data worden doorgegeven.

Governance en traceerbaarheid: Het volledige proces is ontworpen om **transparant en auditbaar** te zijn. Alle prompts voor agents worden extern in files beheerd (geen hard-gecodeerde verborgen prompts), zodat wijzigingen aan agent-gedrag via promptaanpassingen versiebeheerbaar en zichtbaar zijn [9]. Daarnaast wordt elke run van het systeem uitvoerig gelogd: belangrijke metadata zoals run-ID's, gebruikte CLI commando's, configuratiebestanden (bijvoorbeeld `.yaml` modelconfiguraties) en agent-uitkomsten worden vastgelegd in een **sessielog** [10]. Er bestaat bijvoorbeeld een `CODEX_SESSION_LOG.md` waarin chronologisch alle uitgevoerde acties en beslissingen van een run worden geregistreerd [11]. Dit zorgt voor

reproduceerbaarheid: men kan altijd achterhalen welke versie van een prompt of model voor een gegeven output is gebruikt [12]. De content (het digitale kookboek, annotaties, etc.) en logs worden bewaard in een version control repository (git), zodat elke wijziging aan data of procesgeschiedenis wordt getraceerd en er een volledig audit-trail is [13]. Deze focus op documentatie en logging maakt het systeem niet alleen betrouwbaar maar ook overdraagbaar – nieuwe ontwikkelaars of teams kunnen de geschiedenis inzien en begrijpen [14] [15].

Met dit overzicht in gedachten werken we hieronder per aspect de blueprint uit.

Teststrategie en Specificaties

Om zeker te stellen dat het systeem correct functioneert en voldoet aan de requirements, is een uitgebreide teststrategie noodzakelijk. **Voorafgaand aan grootschalige runs moeten alle componenten grondig getest en afgestemd zijn** [16]. We onderscheiden verschillende testniveaus:

- **Unit Tests per component/agent:** Voor elke individuele agent en hulpmodule schrijven we unit tests. Hierbij simuleren we input voor de agent (bv. een stukje recepttekst) en controleren we of de output voldoet aan de verwachtingen (bijvoorbeeld dat de Vertaalagent geen betekenis veranderd heeft, of dat de Annotatie-agent een bepaalde JSON-structuur oplevert). Eventueel kunnen LLM-calls gemockt of gesimuleerd worden met voorspelbare outputs, zodat tests deterministisch zijn. Unit tests garanderen dat de logica van individuele functies en classes correct is.

- **Integratietests (workflow):** Vervolgens testen we de samenhang van agents in de orchestratie. Een integratietest draait bijvoorbeeld de hele pipeline op een kleine batch recepten (of een voorbeeldhoofdstuk) van begin tot eind. We controleren daarbij dat de Orchestrator de agents in de juiste volgorde aanroept, data correct doorgeeft en tussenresultaten opslaat. Belangrijk is om te verifiëren dat de traceerbaarheid intact is – d.w.z. dat voor elke stap logregels en versies worden aangemaakt zoals bedoeld. Ook testen we foutafhandeling: introduceren van een opzettelijke anomaly (bijvoorbeeld twee agents die tegenstrijdige wijzigingen voorstellen) moet resulteren in het juiste governance-gedrag (een **incident stop** door de Orchestrator en aanmaken van een INCIDENT_REPORT voor menselijke interventie, conform ASR-006 regel [4] [17]).

- **Acceptatietests (eind-tot-eind scenario's):** Ten slotte definiëren we per hoofdfunctie acceptatiecriteria en valideren we deze in realistische scenario's. Voorbeeld: *"Als een compleet recept door het systeem gaat, dan moet de eindoutput een vertaling bevatten die inhoudelijk overeenkomt met het origineel, een leesbaar geredigeerde tekst, en minimaal één cultureel-historische voetnoot indien het recept bijzondere ingrediënten bevat."* Dergelijke criteria worden gevalideerd door een run uit te voeren op representatieve recepten en de resultaten door een mens te laten evalueren aan de hand van een checklist. We voorzien acceptatietests voor bijvoorbeeld: **Vertaalkwaliteit**, **Annotatie volledigheid**, **Traceerbaarheidslog** (kan men voor een gegeven zin de *"zo was het — zo is het — en waarom"* terugvinden?), **Performance** (blijft de verwerkingstijd binnen de gestelde grenzen per recept) enz. Pas wanneer alle acceptatietests slagen, beschouwen we de betreffende systeemversie als *"done"* voor productie.

- **Regression tests:** Telkens als nieuwe functies of agents worden toegevoegd (bv. de Methodology Archivist agent of een OCR Integrity agent), draaien we een regressietest suite op eerder verwerkte

content om te verzekeren dat bestaande outputs niet onbedoeld veranderen en dat traceerbaarheidseisen (TR1, TR2, etc.) onveranderd geborgd blijven [18] [19].

Testomgevingen: We hanteren gescheiden omgevingen/configuraties voor tests versus productie. Nieuwe of experimentele agents en workflows worden eerst in een **sandbox modus** getest – hierin draaien agents eventueel in een container of geïsoleerde omgeving zonder schrijfacties op de hoofddata, zodat er geen risico is voor de productiegegevens [20]. Bijvoorbeeld, Phase-3 pilots draaiden in zo'n sandbox met alleen JSON-output en geen directe invloed op canonical data [21]. Pas na beoordeling in de sandbox kunnen changes doorgevoerd worden in de *canonical* workflow. Deze aanpak voorkomt dat ongeteste wijzigingen direct schade aanrichten en zorgt dat tests realistisch maar veilig zijn.

Elke testsoort boven wordt geautomatiseerd zoveel mogelijk (integreren in een CI-pipeline). Ook moeten er duidelijke **acceptatiecriteria per component** gedocumenteerd zijn (bijv. in de Definition of Done van een component staat dat "alle belangrijke beslissingen traceerbaar zijn en risico's niet stilzwijgend genegeerd" worden [22]). Zo is vooraf helder wanneer een onderdeel voldoet.

Deploymentspecificatie (Mac Mini)

Hier beschrijven we hoe het systeem geïnstalleerd en gedeployd wordt op de beoogde hardware, een Apple Mac Mini, in lijn met de technische architectuur.

Voorwaarden & omgeving: De Mac Mini moet beschikken over een recente macOS-versie en internettoegang (voor het ophalen van modellen of het aanroepen van externe API's, indien gebruikt). Idealiter draait hij op Apple Silicon (M1/M2) voor optimale prestaties van lokale ML-modellen. We voorzien een gebruikersaccount "mustikarasa" waarop de services draaien, om af te zonderen van andere gebruikers.

Installatie stappen:

1. **Basis afhankelijkheden:** Installeer Python (versie 3.10+). Dit kan via Homebrew (`brew install python`) of via Xcode Command Line Tools (die Python3 meebrengen). Zorg ook voor Git installatie, omdat de code en data via een git-repository beheerd worden [13].

2. **Code ophalen:** Clone de Mustikarasa repository naar de Mac Mini (bijvoorbeeld `/opt/mustikarasa`). Deze repo bevat zowel de applicatiecode als de markdown-data (boekcontent, logs, prompts, etc.). Dankzij deze **file-based repository** structuur is meteen alle benodigde informatie lokaal aanwezig [23].

3. **Python omgeving:** Creëer een dedicated virtuele omgeving (`python -m venv venv` in de projectfolder) en activeer deze. Installeer vervolgens alle Python-dependencies met pip (`pip install -r requirements.txt`). Dit omvat waarschijnlijk libraries voor OpenAI API toegang (indien gebruikt), lokale LLM runtime aansturing (bv. via Ollama CLI or `crew` scripts), JSON/YAML parsing, etc.

4. **LLM-modellen installeren:**

5. *Indien gebruik van lokale modellen (bv. Mistral via Ollama):* Installeer Ollama op de Mac Mini (via Homebrew: `brew install ollama`). Vervolgens het gewenste model importeren (`ollama pull mistral-7b-v0.1` of een custom model zoals getraind voor dit project). Verifieer dat het model draait door een testprompt via Ollama CLI te sturen.
6. *Indien gebruik van cloud API (OpenAI):* Geen lokale modelinstallatie nodig, maar wel zorgen voor netwerkinstellingen en API-sleutel (zie configuratie). Eventueel een library zoals `openai` installeren.

De architectuur ondersteunt beide opties (lokale of cloud LLM) zonder code-wijzigingen, door de modelkeuze abstract te houden achter de agent-interface [6].

1. **Configuratie (voor runtime):** Plaats de configuratiebestanden op de juiste plek (zie volgende sectie voor details). Denk aan een `.env` file of environment variabelen voor secrets (API keys) en een YAML/JSON config file voor bijv. welk LLM backend actief is, pad naar model, etc. Standaardinstellingen kunnen in de repo aanwezig zijn, maar machine-specifieke settings (zoals API keys, absolute paden) worden hier ingesteld.

2. **Initialiseer data:** De repository bevat al de relevante data (digitale teksten, logs). Mocht er een aparte database of index nodig zijn (lijkt niet het geval, aangezien alles file-based is), dan zouden we die hier opzetten. In deze architectuur is geen zware DB nodig; content wordt direct in Markdown/JSON bestanden beheerd en doorzochte, wat past bij het karakter van een *"knowledge repository"* [13].

3. **Starten van het systeem:** Aangezien het geen traditionele server (met continue listener) is maar een orchestratie die per run wordt uitgevoerd, gebeurt deployment vooral door het beschikbaar maken van tooling. Een run kan bijvoorbeeld gestart worden via een CLI-commando (`python orchestrator.py --batch input/sayur_chapter5` of iets dergelijks). Voor gemak kan men een shell-script of Launchd job schrijven om periodiek of op afroep een run te starten. Als er toch een continue servicecomponent is (bijvoorbeeld een web-API voor status of een scheduler), kan deze in de achtergrond draaien (bijv. als `launchctl` service op macOS of via `nohup`).

4. **Post-deployment checks:** Na installatie draaien we een smoke-test: een kort recept door de pipeline sturen om te kijken of alles werkt (alle agents doorlopen, logs geschreven, output gegenereerd). Ook checken we of alle afhankelijkheden (zoals toegang tot LLM, schrijfrechten op directories, etc.) correct zijn.

Versiebeheer en updates: Omdat zowel code als content in git staan, gebeurt deployment updates door nieuwe commits pullen van de repository. Het is verstandig om op de Mac Mini een geautomatiseerd deploy-script te hebben: `git pull && pip install -r requirements.txt && run tests`. Nieuwe versies doorlopen eerst de testpipeline (lokaal of CI) alvorens in productie (Mac Mini) te worden bijgewerkt, om consistentie te waarborgen.

Configuratiemanagement

Correcte configuratie van het systeem is essentieel, zowel om de juiste parameters te gebruiken als om gevoelige info (zoals API keys) veilig te houden. We hanteren de volgende richtlijnen:

- **Configuratiebestanden:** Gebruik duidelijke, versiebeheerbare config-bestanden voor instelbare parameters. Bijvoorbeeld een `config.yaml` waarin zaken staan als: welke LLM-backend in gebruik is (`backend: "ollama"` of `"openai"`), pad- of URL naar het model, drempelwaarden voor bepaalde agent-acties, toggle voor sandbox mode vs. production mode, etc. Deze YAML-config kan onderdeel van de repository zijn voor standaardwaarden. Eventueel kunnen we meerdere configs hebben per omgeving (bv. `config_prod.yaml` en `config_sandbox.yaml`).

- **Environment variabelen voor secrets:** Gevoelige gegevens zoals API-sleutels (voor OpenAI) of wachtwoorden slaan we niet in git op. In plaats daarvan gebruiken we environment variables of een niet-gecommit `.env`-bestand. Bijvoorbeeld `OPENAI_API_KEY` als env var die door de code wordt gelezen. Op de Mac Mini stellen we deze variabelen in via de shell-profiel of een secure vault (als beschikbaar). Zo staan secrets los van de code en config in repo.

- **Configuratie laden bij runtime:** Tijdens opstart leest de applicatie eerst de basisconfig (yaml) in. Vervolgens worden environment overrides toegepast. Hierdoor kunnen machine-specifieke of secret parameters overschreven worden zonder de algemene config te hoeven aanpassen. Dit maakt het ook eenvoudiger om bijvoorbeeld in een sandbox andere instellingen te gebruiken (bijv. `mode: "sandbox"` in een test-run zodat bepaalde veiligheden aan staan).

- **Traceerbaarheid van configuratie:** Conform de architectuurprincipes wordt elke run ook gelogd met de *effectieve* configuratie. In het sessielog kan bijv. een kopie of verwijzing naar de gebruikte config worden opgeslagen [10]. Dit betekent dat voor elk verwerkingsresultaat later te zien is welke instellingen van kracht waren, wat belangrijk is voor reproduceerbaarheid en debugging [12]. Denk aan: modelversie, promptversies, toggles (wel/geen bepaalde agent ingeschakeld), etc. Dit loggen van config-data gebeurt geautomatiseerd aan begin van een run.

- **Beheer van configuratiewijzigingen:** Omdat configfiles in versiebeheer staan, doorlopen wijzigingen daaraan dezelfde review/governance als codewijzigingen. Bijvoorbeeld, een wijziging in `config.yaml` (zoals verhogen van een throughput-parameter) moet via code review geaccordeerd en gelogd worden. Dit sluit aan bij de eis van transparantie – elke verandering die de werking beïnvloedt is traceerbaar [13]. In de praktijk kunnen we gebruikmaken van Pull Requests of een Changes-log in markdown om configuratiewijzigingen bij te houden.

- **Standaard vs. lokale config:** Hou een **voorbeeldconfiguratie** in de repo (bv. `config.example.yaml`) met gedocumenteerde opties. De daadwerkelijke operationele config (bv. `config.yaml`) kan bij deployment specifiek ingevuld worden. Dit maakt het voor nieuwe implementatoren makkelijk om te zien welke settings er zijn, en voor het systeem om te detecteren indien een config ontbreekt of inconsistent is (zelfs een health-check zou kunnen valideren of alle verwachte configwaarden aanwezig zijn).

Samengevat zorgen we voor een helder scheiding tussen statische configuratie (in files, onder versiebeheer) en dynamische/gevoelige configuratie (in env vars), alles gedocumenteerd en met logging voor audits. Hiermee is de configuratie zowel beheersbaar als flexibel.

Monitoring & Observability

Om het systeem betrouwbaar te kunnen draaien op de Mac Mini, moeten we kunnen **monitoren** wat er gebeurt en snel zicht hebben op eventuele problemen. We richten daarom monitoring en observability in op meerdere niveaus:

- **Logging (Applicatiegericht):** Het systeem genereert al uitgebreide logs per run in Markdown en JSON vorm, die zowel voor mensen leesbaar zijn als door tools parsebaar. We behouden en versterken dit mechanisme. Belangrijke gebeurtenissen, beslissingen en foutmeldingen worden voorzien van gestandaardiseerde logregels [24]. Denk aan een logstructuur met timestamps, componentnaam (agent X, orchestrator), severity level, en message. Cruciale metrics zoals doorvoertijd per agent, aantal wijzigingen aangebracht, etc., kunnen in de log worden opgenomen (of aan het einde van een run een kleine samenvatting). Omdat logs in files staan en onder versiebeheer, kunnen we ze achteraf analyseren en vergelijken tussen runs – dit ondersteunt zowel debugging als evaluatie van kwaliteit [25].

- **Metrics (Prestatiegericht):** Naast textuele logs zetten we waar mogelijk ook kwantitatieve **metrics** uit. Enkele key metrics:

- *Throughput:* aantal recepten per uur dat verwerkt wordt. Dit kan berekend worden door in de log de start- en eindtijd van elke recipe-run te markeren, of een teller bij te houden.
- *Latency:* de tijd die een volledig recept door de hele pipeline doet, maar ook per agent de responstijd. We kunnen de orchestrator laten meten hoelang elke agent-call duurt.
- *Resourcegebruik:* CPU- en geheugenbelasting op de Mac Mini tijdens runs (vooral relevant als lokale modellen draaien). Hiervoor kunnen we systeemeigen tools gebruiken (macOS Activity Monitor of `ps` logging) of lightweight exporters.
- *Fouten:* aantal errors of governance stops per X runs, om stabiliteit te meten.

Deze metrics kunnen simpelweg in een CSV of JSON file bijgehouden worden per run, of via een tool als Prometheus (al is dat misschien overkill lokaal). Een praktische aanpak is een script dat na elke run de logs parseert en een `metrics.md` of dashboard-file bijwerkt met de nieuwste statistieken.

- **Health checks:** Als (toekomstig) onderdeel van een continue dienst kan een health-check endpoint of script nuttig zijn. Bijvoorbeeld een klein HTTP-servicetje dat elke paar minuten rapporteert “up and running” en basisinformatie (vrije schijfruimte, of er een run gaande is, etc.). In de huidige batch-opzet is dit minder cruciaal, maar we kunnen wel een **self-check** script implementeren dat controleert: zijn alle benodigde files aanwezig, is de LLM service (Ollama of API) bereikbaar, kloppen de config settings (bijv. valide API key), etc. Dit script kan voorafgaand aan elke run draaien en bij een misconfiguratie meteen alarm slaan voordat we uren bezig zijn.

- **Dashboards & Alerts:** Voor een enkel Mac Mini systeem is een full-fledged monitoring stack optioneel, maar een eenvoudige dashboard kan nuttig zijn voor ontwikkelaars. We kunnen bijvoorbeeld een lokaal draaiende Grafana instance gebruiken die via een simple JSON datasource

de gegenereerde metrics ophaalt en grafieken toont (doorvoersnelheid over tijd, aantal recepten verwerkt, gemiddelde duur per agent, etc.). Alternatief, een statische pagina of Jupyter Notebook dat de logs inleest en visualisaties maakt na elke run kan ook volstaan. Alerts kunnen eenvoudig worden ingericht via mail of notificaties: b.v. een cronjob die de log parser draait en waarschuwt als er een ERROR of INCIDENT in de laatste run-log staat, of als throughput drastisch daalt. Omdat alles lokaal is, kan men ook de Mac Mini zelf meldingen laten geven (bijvoorbeeld een macOS Notification Center alert bij bepaalde events).

- **Observability culture:** Belangrijk is dat het team regelmatig de logs en metrics **reviewt**. Dit sluit aan bij de audit-requirements: iedere run is een bron van inzicht. Het systeem produceert per fase human-readable artefacten (bv. annotaties in Markdown, proceslogs) die makkelijk te evalueren zijn [26]. We benutten dit door na elke batch-run een kort evaluatiemoment in te bouwen: kloppen de outputs, zijn er opvallende afwijkingen in de metrics, moeten we bijsturen? Deze feedback-loop zorgt dat problemen vroeg ontdekt worden.

Kortom, door een combinatie van gedetailleerde logging [24], het bijhouden van kerncijfers, en eventueel lichte tooling voor visualisatie, houden we vinger aan de pols van het systeem. Dit is essentieel om de kwaliteit en prestaties te waarborgen, vooral als we opschalen naar grotere hoeveelheden recepten of complexere workflows.

Backup & Recovery Procedures

Gegeven het belang van de data (zowel het gedigitaliseerde kookboek als de uitgebreide annotaties en logboeken) is een robuuste backup- en herstelstrategie onmisbaar. We onderscheiden twee categorieën data om veilig te stellen: **sandbox-run data** en **canonical data**.

1. Sandbox runs backup: Sandbox runs zijn testruns of pilot-executies die geen officiële status hebben in het project (ze wijzigen de *canonical* content niet). Toch bevatten ook deze runs waardevolle informatie (bijvoorbeeld evaluaties, experimentele annotaties, debuglogs). We hanteren het principe dat *niets* voorgoed verloren mag gaan tenzij expliciet besloten. Daarom: - **Opslag:** Sandbox outputs (JSON/Markdown) worden doorgaans in een apart pad opgeslagen, bijv. onder `sandbox/` in de repository structuur. Omdat de hele projectrepository onder versiebeheer staat, kunnen zelfs sandbox resultaten daarin worden opgenomen indien gewenst [13]. Als dit ongewenst is (om ruis in git te vermijden), kan men sandbox data ook buiten git houden maar dan periodiek archiveren. - **Backup-frequentie:** We maken regelmatig backups van de sandbox directory, bijvoorbeeld wekelijks of na elke belangrijke pilot. Dit kan in de vorm van het zippen van de sandbox map met een timestamp en kopiëren naar een externe opslag (een NAS, externe schijf of cloud storage). Zo kunnen we altijd een eerdere stand van experimenten terugvinden, zelfs als we lokaal opschonen. - **Opschonen met beleid:** Omdat sandbox data snel kan groeien en niet alles blijvend relevant is, definiëren we een retentiebeleid. Bijvoorbeeld: “bewaar sandbox resultaten minimaal 6 maanden, opschoning daarna alleen na expliciete review”. Belangrijke pilot-runs kunnen gebundeld worden in een *“Pilot Archive”* folder die niet gewist wordt.

2. Canonical data backup: De canonical data omvat de gevalideerde, officiële projectresultaten: de geautoriseerde digitale tekst van Mustika Rasa (in markdown/XML), de definitieve annotaties, woordenlijst, en de audit-trail (sessielogs, beslislogs). Deze data is uiterst waardevol en moet met de hoogste prioriteit beveiligd worden tegen verlies. Onze aanpak: - **Version control als eerste lijn:** Omdat alle content en wijzigingen in een git-repository staan, fungeert dit al als een vorm van backup [13]. Elke commit houdt een

stukje geschiedenis vast. We zorgen ervoor dat de repository op een **remote server** wordt gespiegeld (bijv. GitHub of een privé Git-server). Zo is er minimaal één offsite kopie van alle versies. Push na elke serie wijzigingen de commits naar de remote om geen lokale veranderingen ongebackuped te laten. - **Full backups:** Naast git (dat vooral incrementele tekstgeschiedenis dekt) maken we periodiek volledige snapshots van de repository en relevante directories. Bijvoorbeeld elke nacht een backup job die de gehele projectfolder (inclusief evt. niet-geversioneerde bestanden zoals geïnstalleerde modellen of bulky assets) archiveert. Deze backup kan versleuteld worden en opgeslagen op cloud storage of een fysiek offsite medium. Mac Mini's Time Machine kan hiervoor ook worden gebruikt als aanvullende maatregel (het vangt systeem breed wijzigingen). - **Database/filesystem dumps:** Mocht er toch gebruik zijn van additionele storage (stel in de toekomst een database voor bepaalde metadata), dan scripten we ook daar regelmatige dumps/exporten van. In huidige architectuur is dit niet van toepassing doordat we file-based werken [23]. - **Backup verificatie:** Maandelijks wordt een integriteitscheck gedaan: kunnen we een backup teruglezen? We simuleren bijvoorbeeld op een aparte machine het terugzetten van de backup om te verifiëren dat alle cruciale bestanden intact zijn en het systeem vanuit die backup zou kunnen draaien. - **Continuïteitsplanning:** We documenteren expliciet de procedure om vanaf een backup te herstellen: stap 1) een schone machine voorbereiden, 2) repo/backup terugplaatsen, 3) omgeving opzetten, etc. Dit document helpt in een noodsituatie (bv. hardware defect Mac Mini) snel weer operationeel te zijn. Ook benoemen we wie verantwoordelijk is voor het initiëren van recovery en hoe te communiceren als data loss dreigt (*incident response* plan).

Met bovenstaande maatregelen zorgen we dat zowel experimentele data als officieel gecureerde data veiliggesteld zijn. In het onwaarschijnlijke geval van corruptie of verlies kunnen we snel terug naar een consistente vorige toestand. Bovendien ondersteunen de interne systeemfeatures dit: de applicatie zelf heeft een undo/rollback mechanisme voor batchedit acties [27], wat inhoudt dat, mocht een fout doorgevoerd worden in canonical content, we via zowel backups als via het systeem zelf wijzigingen kunnen terugdraaien zonder gegevensverlies.

Security Specificatie

Beveiliging is cruciaal, ondanks dat dit project voornamelijk om publieke historische tekst gaat. We willen ongeoorloofde toegang, datamanipulatie en uitlekken van gevoelige informatie (zoals API keys) voorkomen. De security-aanpak omvat:

- **Besturingssysteem & Toegang:** De Mac Mini wordt up-to-date gehouden met macOS security patches. We maken slechts accounts aan voor geautoriseerde personen/systemen. Waar mogelijk gebruiken we SSH-toegang met sleutelauthenticatie in plaats van wachtwoorden, om de machine remote te beheren. Physical access tot de Mac Mini is beperkt tot de eigenaar/beheerder. De firewall staat aan en blokkeert inkomende verbindingen behalve wat expliciet nodig is (mogelijk is er helemaal geen open poort naar buiten, aangezien runs lokaal gestart worden).

- **Bestandspermissies:** Alle projectbestanden (repository) staan onder een dedicated gebruikersaccount (bijv. `mustikarasa` user) en zijn niet toegankelijk voor andere gebruikers op het systeem. We zetten umask of permissies zo dat nieuwe logfiles of output niet wereld-leesbaar zijn. Backups die extern worden geplaatst, worden versleuteld opgeslagen zodat gevoelige details (bijv. discussies in logs) niet door derden ingezien kunnen worden.

- **Geheimenbeheer:** Zoals eerder genoemd, API-sleutels en andere secrets worden nooit in plaintext in de code of repo opgenomen. Ze verblijven als environment variabelen of in een lokaal secure vault (op macOS zou bijvoorbeeld de ingebouwde Keychain gebruikt kunnen worden voor opslag van credentials). Toegang tot deze secrets is alleen mogelijk voor de applicatie zelf en de systeembeheerder. In de codebasis vermijden we ook het per ongeluk loggen van secrets (dus geen debug prints van de API key bijvoorbeeld). Als we CI/CD gebruiken voor het project, worden secrets daar in encrypted variables beheerd.

- **Integriteitsbescherming content:** Hoewel de content geen persoonlijke data is, hechten we veel belang aan integriteit (het kookboek moet authentiek blijven). Daarom zijn er technische en procesmatige controles om ongeautoriseerde of onbedoelde wijzigingen te voorkomen [28]. Technisch betekent dit: alleen via de goedgekeurde workflow (orchestrator + human gate) kunnen veranderingen doorgevoerd worden in de canonical dataset. De repository kan beschermd worden met branch-protectie – directe commits op de main branch (canonical data) alleen via gereviewde pull requests, zodat altijd een tweede paar ogen meekijkt bij wijzigingen. Tevens draait er een governance-agent/triggers die verdachte wijzigingen of policy-overtredingen detecteert en stopt [29], bijvoorbeeld als een agent zou proberen een verboden actie uit te voeren wordt de pipeline gepauzeerd voor menselijk ingrijpen.

- **Externe AI-diensten:** Indien gebruik gemaakt wordt van cloud LLMs (OpenAI), zorgen we ervoor dat we **geen gevoelige data** naar externe servers sturen [28]. In dit geval betreft de data historische recepten – niet privé, maar we letten erop dat bijvoorbeeld API calls geen verborgen extra info meesturen. We kunnen een content-filter toepassen om zeker te zijn dat er geen per abuis vertrouwelijke notities of zo tussen zitten. Ook wordt de communicatie met de API via HTTPS gedaan (standaard voor OpenAI API). De API keys worden met de voorgenoemde voorzorg bewaard en in requests nooit in logfiles opgeslagen. Als we lokale modellen gebruiken, vermijden we deze risico's helemaal, maar dan is beveiliging van het model op de machine zelf van belang (zorgen dat niet iemand stiekem de model-run hijackt om eigen prompts te sturen).

- **Rechten van processen en diensten:** We hanteren het principe van **least privilege**. De orchestrator draait onder een gewone gebruiker (niet als root), en heeft slechts schrijfrechten in de projectdirectories waar dat nodig is. Er worden geen onnodige netwerk services gedraaid op de Mac Mini. Als we containerization inzetten voor sandbox tests, hebben die containers beperkte toegang tot de host (bijv. geen toegang tot internet of host filesystem buiten hun sandbox-folder, om risico's van eventuele malafide modelcode te beperken).

- **Auditing en monitoring voor security:** Naast functionele monitoring, kijken we ook naar beveiligingsrelevante logs. Pogingen tot ongeautoriseerde toegang (bijv. een onbekende SSH login) worden gelogd en gemeld. Changes in belangrijke configbestanden of binaries op het systeem kunnen gemonitord worden (Host Intrusion Detection, bv. tripwire of simpelweg regelmatig diff'en van kritieke bestanden). Omdat alle bewerkingen in het systeem ook gelogd worden per run, is er een **volledig audit trail** van de inhoudelijke wijzigingen [13]. We kunnen periodiek een audit uitvoeren op deze logs om er zeker van te zijn dat alle wijzigingen gerechtvaardigd en zoals verwacht zijn doorgevoerd (dit is deels ook een kwaliteitseis, maar sluit aan op security: het detecteren van anomalieën of sabotage).

Concluderend richten we de security zo in dat zowel de **omgeving** (Mac Mini, bestanden) als de **applicatieprocessen** beveiligd zijn. Het project volgt daarmee de best practices: restrictieve toegang, traceerbaarheid van acties, geen blootstelling van gevoelige gegevens extern, en controles op de integriteit van het proces [30] [31].

Performance Requirements & Optimalisatie

Hoewel de initiële focus lag op nauwkeurigheid en traceerbaarheid boven snelheid, is het belangrijk om prestaties te specificeren en waar mogelijk te optimaliseren. Hieronder de performance-eisen en hoe we deze halen:

- **Throughput (recepten per uur):** Het systeem moet een bepaalde doorvoersnelheid halen zodat het project binnen redelijke tijd resultaten kan opleveren. Op de huidige hardware mikken we bijvoorbeeld op **minimaal 5 recepten per uur** bij volledige verwerking. Deze target is gebaseerd op een inschatting van ~12 minuten per recept, wat haalbaar lijkt met een lokale 7B LLM of API-calls, inclusief alle agents. Indien we dit niet halen, moeten we schalen (zie verder). We monitoren continu de feitelijke throughput metric; als deze onder de doelwaarde zakt, is dat een signaal om de oorzaken te onderzoeken (bijv. een agent die te traag is).

- **Latency (reactietijd per component):** Naast totale doorlooptijd meten we hoe lang elke agent nodig heeft voor zijn taak. We stellen eisen aan maximale latencies: bijv. een vertaalactie mag max 2 minuten duren, een annotatie-agent 1 minuut, etc., anders wordt het geheel te traag. Als een agent structureel boven zijn budget zit, bekijken we optimalisaties (prompt vereenvoudigen, model wisselen, of resultaat cachen). Een end-to-end run van één recept zou idealiter niet langer dan 10-15 minuten duren voor een soepel proces, maar dit is flexibel afhankelijk van complexiteit van het recept.

- **Resourcegebruik:** De Mac Mini heeft begrensde CPU/GPU en geheugen. We specificeren daarom dat tijdens processing CPU gebruik liefst onder 80% gem, zodat de machine responsief blijft en thermische throttling voorkomt. Geheugengebruik (RAM) door de Python process en geladen model samen moet onder, zeg, 12 GB blijven (aannames voor een 16GB machine) om swapping te voorkomen. Deze limieten zijn richtlijnen; we meten daadwerkelijk gebruik tijdens tests. Als een lokale model teveel geheugen eist, overwegen we een kleinere model of optimalisatie (bijv 4-bit quantization van het model).

- **Schaalbaarheid & Parallelisme:** De architectuur is schaalbaar opgezet zodat we throughput kunnen verhogen wanneer nodig [32] [14]. Enkele strategieën:

- *Batch-parallelisatie:* In plaats van recepten strikt sequentieel één voor één te verwerken, kunnen we meerdere recepten parallel door de pipeline laten gaan, zolang ze onafhankelijk zijn. De orchestrator kan bijvoorbeeld opgestart worden in meerdere aparte processen/threads, elk met een eigen content batch. Omdat sommige agents (zoals Continuity/Cohesion-checks) mogelijk batch-breed werken, moeten we wel oppassen dat parallel runs elkaar niet hinderen. Maar bijvoorbeeld het verwerken van **verschillende hoofdstukken parallel** is goed mogelijk. Dit was al voorzien in het ontwerp (conceptualiseren van parallel workflows) [32].
- *Asynchroon binnen agents:* Waar mogelijk maken we agent-calls asynchroon. Bijvoorbeeld, als een agent externe API calls doet of I/O, kan de orchestrator intussen misschien alvast een andere taak

voorbereiden. In Python zouden we `asyncio` of multi-threading kunnen inzetten per agent voor overlappende I/O.

- *Opschalen hardware:* Mocht de Mac Mini niet volstaan qua rekenkracht, is verhuizen naar zwaardere infrastructuur een optie. Door de inzet van een generieke Python codebase zonder platform-specifieke hacks kan het systeem relatief eenvoudig op een Linux server of in de cloud draaien [7]. Denk aan een VM met meer CPU-cores of een GPU voor versnelling. Dankzij containerisatie van sandbox-tests is het mogelijk de hele runtime in een container image te gieten en op een nieuwe host uit te rollen.
- *Model- en promptoptimalisatie:* Een belangrijke hefboom voor performance is het AI-model zelf. De architectuur laat toe een ander model te kiezen zonder codewijziging [6]. We kunnen bijvoorbeeld overschakelen naar een snellere LLM als throughput omhoog moet, zij het mogelijk ten koste van kwaliteit. Een concreet plan: gebruik Mistral-7B voor snelle iteraties, en alleen voor kritieke eindcontroles een zwaarder model. Ook promptoptimalisatie kan latency drastisch verlagen – kortere prompts of gerichtere vragen leiden soms tot snellere responses van het LLM. Dit moet gebeuren zonder de inhoudelijke juistheid te schaden.

- *Caching:* Overweeg caching van resultaten voor onderdelen die herhaaldelijk dezelfde output geven. Stel dat bepaalde standaardzinnen of veelvoorkomende terminologie steeds door een agent op dezelfde manier verwerkt worden, dan zou een cache hits kunnen opleveren (alhoewel door de creatieve aard van LLM output dit beperkt bruikbaar is). Wel nuttig is het cachen van externe bronnen (bijv. als een agent een API call naar een woordenboek doet voor elk recept, cache die resultaten).

- **Testen & Bijsturen van performance:** Performance requirements worden meegenomen in de testen. We creëren bijvoorbeeld een performance test die 5 recepten verwerkt en verifieert dat het binnen 1 uur gebeurt. Ook zetten we alarms in monitoring als throughput onder een drempel zakt. De resultaten geven we terug aan ontwikkelaars: als een agent de bottleneck blijkt, kan die wellicht herzien worden (bijv. een lichtere model toepassen voor die stap). Het ontwerp is modulair genoeg dat we hardware- of modelupgrades kunnen “inpluggen” om throughput te verbeteren [33]. Dit betekent concreet dat we bijvoorbeeld een tweede Mac of cloud-VM kunnen toevoegen die parallel draait, of een nieuwere generatie LLM integreren voor snellere resultaten, zonder de architectuur te hoeven hertekenen.

Samengevat leggen we duidelijk vast welke prestaties we minimaal verwachten en houden we het systeem zo flexibel dat we kunnen opschalen. Dankzij de schaalbaarheid en modulariteit van het ontwerp kan de capaciteit vergroot worden door batch-parallelisatie en betere hardware, zodra de nodig is [32] [34]. Zo kunnen we toekomstige groei in het aantal recepten of projecten aan zonder kwaliteitsverlies.

Migratie- en Upgradepad

De architectuur moet niet alleen in het heden werken, maar ook **toekomstbestendig** zijn. We beschrijven daarom hoe upgrades van versie 1 naar versie 2 (en verder) uitgevoerd kunnen worden **zonder dataverlies** en met minimale onderbreking. Het project zit immers in fases (scholarly edition naar public edition, enz.), dus evolutie is onderdeel van de planning.

Software-versie upgrade: Wanneer er een nieuwe versie van de Mustikarasa-software komt (bijvoorbeeld van v1.0 naar v2.0), volgen we een gecontroleerd deploymentproces: - In een staging-omgeving (eventueel

de sandboxmodus op dezelfde Mac Mini of een aparte machine) installeren we de nieuwe versie naast de oude. We draaien alle **unit- en integratietests** om te verzekeren dat v2.0 compatibel is met de bestaande data en geen regressies introduceert. - We voeren een migratiescript uit als dat nodig is voor data. Bijvoorbeeld, als v2 een iets ander formaat voor logs of recepten gebruikt, schrijven we een script dat de bestaande markdown/JSON omzet naar het nieuwe format. Dit script wordt eerst in staging getest op een kopie van de data. - Pas daarna schakelen we productie om: we stoppen het systeem (zodat er geen runs actief zijn), maken een laatste backup van de v1 toestand, upgraden de code naar v2 (git pull of nieuwe container deployen), voeren het migratiescript op de **canonical data** uit, en starten dan het systeem weer op in v2 modus. Omdat we eerst een backup hebben, is er rollback mogelijk als er onverwacht toch iets misgaat.

Datamigratie en compatibiliteit: Een belangrijk principe is **non-destructiveness**: bestaande canonical data wordt bij voorkeur niet gewijzigd door een upgrade, of alleen op een toevoegende/manipulatieve wijze zonder informatieverlies. Mocht in v2 bijvoorbeeld de representatie van bepaalde annotaties veranderen, dan converteren we oude annotaties naar het nieuwe schema, maar bewaren we indien mogelijk ook de oude versie ergens (desnoods in git history of in een archief) voor naslag. In het ideale geval is v2 zo ontworpen dat hij oudere data nog kan lezen/gebruiken. Bijvoorbeeld: als de logs van v1 in Markdown zijn en v2 stapt over op JSON logs, dan zou v2 een backwards-compatibiliteitsmodus kunnen hebben om oude Markdown logs te parse voor continuïteit.

Geleidelijke overgang: Indien het risico of de impact groot is, kunnen we kiezen voor een **gefaseerde migratie**. Dat wil zeggen, draai v2 initieel voor nieuwe content, maar laat v1 voorlopig verantwoordelijk voor al verwerkte content. In de praktijk: content die al canonical is blijft in v1 formaat, nieuwe runs gebeuren met v2 (en nieuwe structuur), en we ontwikkelen een bridging-tool die zorgt dat beide netjes naast elkaar getoond kunnen worden in eindoutput of UI. Gaandeweg kunnen we dan oudere content door v2 halen voor uniformiteit. Deze aanpak voorkomt een big-bang migratie en maakt terugrollen eenvoudiger (men kan dan immers oude content onaangeroerd laten als fallback).

Upgraden zonder data loss: Dankzij strikte versiecontrole van alle content [13], is er altijd een historisch record. Bij upgrade gaat niets verloren: zelfs als data gemigreerd wordt, blijft de oude versie in git history beschikbaar. Toch willen we dat gebruikers in de nieuwe versie niets missen. Daarom valideren we na migratie dat het **canonieke eindresultaat identiek** is gebleven waar het moet. Bijvoorbeeld, random check: open een recept van voor de upgrade en na de upgrade, er mag inhoudelijk geen verschil zijn behalve toegestane format-aanpassingen. Onze acceptance tests kunnen dit deels automatiseren.

Schema en configuratie veranderingen: Nieuwe versies kunnen veranderingen in config vereisen. We documenteren helder welke nieuwe config-opties erbij zijn gekomen of oude vervangen. Het migratieproces omvat het bijwerken van config files (semi-automatisch via scripts of handmatig volgens handleiding). Omdat settings ook onder versiebeheer zijn, kunnen we de diff bekijken en peer-reviewen – zodat we zeker weten dat bv. een toggle `ALLOW_AUTO_GLOSSARY` in v1 die verdween in v2 correct is opgevolgd door nieuwe logica elders.

Content-uitbreiding en nieuwe functies: Sommige upgrades zijn niet alleen technisch, maar functioneel – bijvoorbeeld de toevoeging van Phase 2 (publieke editie) functionaliteit. De architectuur is van meet af aan ontworpen om uitbreidbaar te zijn met nieuwe agents en fases zonder herbouw van nul [35]. Concreet betekent dit dat het toevoegen van een nieuwe fase een configuratiekwestie is (nieuwe agent pipelines toevoegen) plus natuurlijk nieuwe promptfiles en code voor die agents. Maar het bestaande systeem blijft

draaien voor Phase 1 zoals het is. We kunnen Phase 2 features *enable* per config als ze klaar zijn, dus er is een soort feature toggle mogelijk om de overgang soepel te maken. Bijvoorbeeld, v2 bevat al code voor de **Design Agent** en **Public Edition adjustments**, maar we zetten die pas aan (via config) nadat v1 content gemigreerd is en team akkoord geeft. Dit voorkomt data loss doordat we niet halverwege de content ineens alles anders gaan doen; het gebeurt gepland en gecontroleerd.

Rollback plan: Geen migratie is compleet zonder rollback-mogelijkheid. Zoals genoemd houden we v1 backups paraat. Mocht er na livegang van v2 een ernstig probleem blijken, kunnen we terug naar v1 door de oude omgeving terug te zetten en de data vanuit backup/git herstellen. Omdat we in de migratie niets verwijderd hebben, is dit relatief eenvoudig: het zou hooguit betekenen dat wijzigingen in v2 nog eens opnieuw moeten gebeuren in v1 of worden verrekend. Het systeem's eigen logging en versiebeheer helpen ook hier: we kunnen precies zien wat v2 eventueel aan content veranderd heeft en dat gericht ongedaan maken als nodig.

Documentatie en communicatie: Elke migratie/upgradestap wordt gedocumenteerd in een migratiehandleiding. Bovendien wordt alle relevante belanghebbenden (ontwikkelaars, eventueel eindgebruikers indien van toepassing) gecommuniceerd wat er verandert. Intern houden we changelogs bij. Dit sluit aan bij de transparantiecultuur: net zoals elke agent-beslissing gemotiveerd is, documenteren we ook elke grote systeemwijziging inclusief rationale ("waarom upgraden, wat verandert er functioneel") en gevolgen.

Met dit migratiepad is het systeem klaar om **veilig te evolueren**. De huidige architectuur biedt al veel flexibiliteit zodat grote aanpassingen mogelijk zijn zonder alles te herzien [35]. Deze blueprint zorgt dat bij zulke aanpassingen de continuïteit van data en werking gegarandeerd blijft. Het ontwerp heeft een flinke "runway" om door te groeien zonder van de visie af te wijken [35] – vergelijkbaar met een vliegtuig dat kan opstijgen zodra alle checks gedaan zijn, zal Mustikarasa v2 vlekkeloos van de grond komen op basis van de solide fundering van v1.

---

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 Mustikarasa System Architecture Overview (TOGAF-Based Approach).pdf
file://file_00000000025071f4a6714b3d3c5b6a2e