ChatGPT

Phase 2 Implementation Plan: AI-Driven Longevity Coach Website on GCP

Overview and Objectives

Phase 2 builds on the Phase 1 infrastructure (GKE cluster, CI/CD, IAM setup) to deliver the first fully functional website with **dynamic AI-generated content**. This phase implements the core web application and the AI content pipeline, aligning with the project's goals of up-to-date longevity content, a clean web interface, and robust GCP integration [1]. We will use Google Cloud services (Firestore, GKE, Vertex AI) alongside AI frameworks (LangChain, LangGraph, Anthropic's MCP) to achieve a maintainable, scalable system [2] [3]. The plan is broken into milestones that are modular and logically ordered, ensuring each piece is built and tested before moving to the next.

Milestone 1: Establish Firestore-Backed Content System

Goal: Set up a cloud database to store website content (articles, tips, etc.), enabling dynamic content delivery. We choose **Google Firestore** for its flexibility and seamless integration with GCP. Each piece of content (e.g. an article) will be a document in a Firestore collection, with fields for title, body, date, tags, etc. [4]. This schema will allow querying by attributes (e.g. latest articles, or by topic) and easy updates by our AI agents.

Implementation Tasks:
- **Enable Firestore in GCP:** If not already enabled in Phase 1, use Terraform or gcloud to activate the Firestore API in the project. Choose the appropriate mode (Native mode Firestore for new projects).
- **Create Collections:** Define a Firestore `articles` collection for long-form content and other collections like `tips` for short tips. Each article document will store fields such as `title`, `content_body`, `summary`, `published_date`, and perhaps a list of `source_urls` or `tags` [4].
- **Service Account & Security:** Extend IAM so that a GCP service account (for the web app and agents) has permissions to read/write Firestore documents in these collections and nothing more. This follows the least-privilege principle (e.g. a role like *Cloud Datastore User* scoped to the needed collection) [2]. Ensure this access is via Workload Identity (no plaintext keys) for secure access from GKE pods.
- **Data Access Layer:** In the application code, implement a simple data access module using the `google-cloud-firestore` Python client [4]. This module will provide functions like `get_latest_articles()`, `get_article(id)`, `save_article(data)` etc., abstracting Firestore calls.
- **Seed Content for Testing:** Insert a few sample documents (manually or via a script) to verify that the Firestore setup works. For example, add a placeholder article and tip so the frontend can display something before the AI pipeline is live.

Milestone 2: Develop Basic Frontend with FastAPI

Goal: Create the web frontend that users will interact with, served by a Python FastAPI application. In this phase, we build a minimal but functional website structure – pages for viewing articles, listing tips, etc. – using static or placeholder content initially [5]. This establishes the UI/UX framework so dynamic content can be plugged in later. FastAPI is chosen for its performance and easy integration of both

HTTP routes and WebSocket (for future interactive coach) [6]. We will keep the design clean and responsive, using a simple CSS framework for quick styling [7].

Implementation Tasks:
- **FastAPI App Scaffolding:** Initialize a FastAPI project (if not already done in Phase 1 "Hello World"). Set up routing structure: e.g. a route for the home page (`/`), an articles page (`/articles` listing summaries), an article detail page (`/articles/{id}`), and a placeholder page for Q&A (to be implemented in a later phase). Use Jinja2 templates for server-side rendering of HTML pages [6], so content can be embedded easily.
- **Basic Templates & UI:** Implement minimal HTML templates. Integrate a CSS framework like **Bootstrap** or **Tailwind** to ensure a clean, mobile-friendly design without heavy custom CSS [7]. The navigation bar should allow users to switch between "Articles", "Tips", and the future "Ask the Coach" section. At this stage, content displayed can be sample text or data from Firestore seed documents.
- **Firestore Integration (Read):** Connect the FastAPI backend to Firestore via the data layer from Milestone 1. For example, the `/articles` route handler calls `get_latest_articles()` to retrieve articles from Firestore and passes them to the template for rendering. Similarly, a `/tips` route can show the latest tip. This proves that the web app can read dynamic content from the database.
- **Testing the Web UI:** Run the FastAPI app locally (or on the dev cluster) and verify that all pages load and display the placeholder content. Ensure navigation links work. This is still a static content phase (no AI yet) but confirms the web interface and routing are functional [8]. We should see a basic site where users can click through sections, even if the content is just sample data.

Milestone 3: Implement the AI Content Generation Pipeline (Multi-Agent System)

Goal: Develop the **multi-agent pipeline** that will automatically generate new longevity content. This pipeline consists of four coordinated agents – Aggregator, Summarizer, Editor, and Publisher – each with a specific role as described in the project plan. We will use **LangChain** to implement each agent's LLM logic and tool usage [9], and leverage **LangGraph** (a LangChain extension) or simple orchestration code to chain the agents into a workflow [10]. Agents will utilize Anthropic's **MCP** for safe tool integration (e.g. web searches) so the LLMs can fetch external data securely [11] [12]. In this milestone, the focus is on coding each agent and testing the pipeline end-to-end in a development environment (offline or non-production) with dummy triggers.

Implementation Tasks:
- **Aggregator Agent (Researcher):** Develop the first agent to fetch raw information on longevity. This agent will run periodically (triggered by a scheduler in a later milestone) and gather recent content from external sources [13]. Use LangChain to give the agent the ability to call tools – for example, an HTTP tool to fetch RSS feeds or an API wrapper for news sites. We will likely implement a custom MCP tool server for web search or RSS feed retrieval [11] [12], ensuring the LLM can ask the tool for data without uncontrolled code execution. The agent's prompt will instruct it to find *recent, relevant* longevity news (e.g. "Find the latest research or news in longevity from the past day"). It should output a collection of raw text or links (e.g. titles and summaries of two or three new articles).
- **Summarizer Agent (Writer):** Develop the second agent to take the Aggregator's findings and produce a draft article or summary. This will use a Large Language Model via LangChain to **summarize key points** from the collected info [14]. For now, use a Vertex AI text generation model (or OpenAI GPT-4 via API) to implement this agent. The prompt might be along the lines of: *"Write a concise, informative article in a friendly coaching tone about the following research findings…"*, then insert the aggregator's data. The agent should output a structured draft (with a suggested title, body text, maybe bullet points or Q&A if appropriate). We will also prompt it to keep within a certain length and maintain an engaging tone (per

content guidelines).
- **Editor Agent (Reviewer):** Develop the third agent to review and refine the draft from the Summarizer. This agent is essentially a proofreading and fact-checking layer [15] [16]. It can be implemented as another LLM call (e.g., prompt the model with the draft text and ask for improvements). Key tasks for the Editor agent: fix grammar or clarity issues, ensure the tone matches our coach persona, check for any factual inconsistencies, and possibly insert a citation or disclaimer for any scientific claims [17] [18]. For example, if the summarizer wrote "Studies show X extends life," the editor could clarify which study or add "according to a 2025 study...". The output of this agent is a polished final article ready to publish.
- **Publisher Agent (Publisher):** Develop the final agent responsible for publishing the content to the site. We have two possible approaches as outlined in the plan: **(a) Direct publish to Firestore** or **(b) Content-as-Code via Git commit** [19] [20]. To meet our Phase 2 objectives and demonstrate CI/CD integration, we will implement approach (b) initially (treat content as code). The Publisher agent will take the final article (title + body from Editor) and create a Markdown or JSON file representing that article. Using GitHub's API, it will commit this file to a designated content folder in the repository (or open a Pull Request) [20]. This means every new article goes through version control, with history tracked. (In a later iteration, we could also support direct Firestore publishing for instant updates, but the content-as-code method is useful for now to leverage our GitOps workflow [21].)
- **Orchestration & Testing:** Tie the four agents together in a pipeline. Using LangChain's chaining or a LangGraph workflow, define the sequence: **Aggregator -> Summarizer -> Editor -> Publisher** [10]. The data output of one feeds into the next. We will implement this as a Python function or script that calls each agent in order, handling data hand-off. Include error handling: if any agent fails (e.g., no data found or an API error), catch exceptions so the whole pipeline doesn't crash ungracefully [22]. Once implemented, **test the pipeline offline**: run the chain manually (maybe via a CLI command or Jupyter notebook) to see it produce an example article from start to finish [23]. This may take several prompt tuning iterations to get coherent output. We will verify that at the end of the run, the Publisher either committed a file (if using content-as-code) or at least produced the final content object. This milestone is successful when we can generate a reasonable "longevity news summary" article through the pipeline in a controlled test.

Milestone 4: Integrate the AI Pipeline with the Web Application

Goal: Connect the content generation pipeline into the live website environment so that AI-generated articles actually appear on the site dynamically. This involves feeding the pipeline's outputs into our Firestore content system (or repository) and enabling the frontend to display new content. We will also deploy the pipeline to run on a schedule (via Kubernetes CronJob) so content updates occur automatically. By the end of this milestone, the site will **update itself with fresh longevity content** (e.g. daily) without human intervention [24].

Implementation Tasks:
- **Content Storage Integration:** Decide on the publishing path and implement it in production. If using the **Firestore direct approach**, have the Publisher agent call the Firestore data layer to save the new article document (with all relevant fields). The FastAPI frontend will then automatically include it (since it already reads from Firestore). If using **content-as-code (GitHub commit)**, set up the repository structure and CI logic so that when an article Markdown is added, the site will include it. For example, the FastAPI app could load articles from a local folder (populated by git), or we have a step in the CI/CD that takes the committed content file and writes it to Firestore. We might implement a hybrid: use the Git commit approach to practice the AI-to-PR workflow, but then have the deployment process actually update the Firestore or static files on the web server with that content [20] [21]. In any case, ensure that a newly generated article ends up accessible via the `/articles` page after the pipeline runs.
- **Frontend Dynamic Content:** Modify the FastAPI routes and templates to display real content from the database. For instance, the articles listing page should query Firestore for the most recent articles and

render them (title, date, snippet). If we have article detail pages, implement a route that takes an article ID and fetches the full content for display. Verify that after the Publisher agent runs and stores a new article, it shows up on the site (e.g. appear on the "Latest Articles" list) [19]. This likely involves adding a timestamp field so we can sort by newest. Also implement basic pagination or limiting (e.g. show the latest 5 articles).
- **Kubernetes CronJob for Scheduling:** Set up a **CronJob** resource in the GKE cluster to run the content generation pipeline on a schedule (e.g. once per day, perhaps late at night or early morning). The CronJob will use a Kubernetes Job specification that runs our pipeline code. We can reuse the existing Docker image for the app: for example, add an entrypoint/command that triggers the pipeline (such as `python run_pipeline.py`). The CronJob YAML will reference this image and command, scheduled at the desired interval [25]. Use `kubectl` (or Terraform if managing infra as code) to create the CronJob. Test it by manually creating a Job from the same spec to ensure the pipeline runs correctly in-cluster. Monitor logs to see that it goes through all agent steps and either commits content or writes to Firestore.
- **Idempotence and Duplicate Prevention:** Incorporate logic to avoid posting duplicate content if the CronJob runs multiple times or if data hasn't changed [26]. For example, the Aggregator agent can keep track of the IDs of news items it has seen (store in Firestore or memory) and skip those. Or the Publisher can check if an article with the same title already exists in the database before creating a new entry. This prevents spam or repeated posts in case the same news appears on consecutive days. We might implement a simple check like storing a hash of source URLs of the last run. Ensuring idempotence is important so the pipeline can be safely re-run as needed.
- **Deploy and Verify Dynamic Content:** With scheduling in place, let the pipeline run and populate the site. After the first scheduled run (or manual trigger), verify the website now shows a **new AI-generated article** in the articles section. This confirms end-to-end integration. At this point, the site is dynamic – content updates happen automatically – fulfilling the core promise of the AI-driven coach. We can consider Phase 2 "feature complete" here: the public can see fresh longevity content generated by our AI agents.

Milestone 5: Continuous Integration & Deployment (CI/CD) Enhancements for Code and Content

Goal: Leverage and extend the Phase 1 CI/CD pipeline to streamline Phase 2 development and to incorporate the **content-as-code workflow**. We ensure that any change – whether application code or AI-generated content – goes through our automated build/test/deploy process on GitHub. This milestone focuses on integrating content updates into CI/CD and enforcing good DevOps practices (testing, containerization, version control for everything).

Implementation Tasks:
- **GitHub Actions Workflow for Phase 2:** Update the existing GitHub Actions workflows to account for the new components. For example, extend CI steps to run **unit tests for the agent logic** (e.g. test the aggregator parsing, a dry-run of summarizer on sample input) [27]. Ensure that when a Pull Request is opened (which could be code changes or an AI content PR), the CI runs all tests and also builds the Docker image. We will use a **multi-stage Dockerfile** to build the app, which now includes FastAPI + agents code, keeping the final image slim and secure [28]. If content files (Markdown) are part of the repo, we might add a validation step (e.g. check formatting or frontmatter) in CI for those as well.
- **Automated Deployment:** Confirm the CD pipeline deploys the updated application to GKE on each merge to main. This was set up in Phase 1, but now the presence of new content files in the repo should also trigger a deployment so the site updates. The GitHub Actions CD job will push the new Docker image to Artifact Registry and update the Kubernetes Deployment on GKE (e.g. using `kubectl set image` or Helm upgrade) [29] [30]. We expect zero-downtime rolling updates as per K8s best practices.

For content-only changes (like a PR that only adds an article file), rebuilding the container might not strictly be necessary if content is decoupled, but our pipeline will treat it the same – ensuring the live site always reflects the latest main branch state.
- **Content-as-Code Integration:** Since we opted to treat content updates via Git, set up a pattern where the **Publisher agent opens a Pull Request** with new content. That PR will undergo the same CI checks and then can be merged (possibly automatically if tests pass) to publish the content [20]. We can even automate the merge if we trust the pipeline output, or have a maintainer quickly review the AI-written content. Once merged, the CI/CD deploys it. This demonstrates an AI-in-the-loop development cycle: AI writes content, goes through version control, and our infrastructure deploys it, providing traceability for all changes [31].
- **Monitoring CI/CD:** Configure notifications (email/Slack) for pipeline results so we know when deployments succeed or if something fails. Also, ensure sensitive data (like any API keys for external tools) are stored in GitHub Actions secrets and referenced securely (for example, if the aggregator uses an API key for a news API, store it as a secret and inject as env var during the job).
- **Containerization Best Practices:** As part of CI, verify the Docker image for security and efficiency. Use linters or scanners (like Dockerfile lint, Trivy) to catch vulnerabilities. The Dockerfile will run the app as a non-root user for security, and only include necessary runtime dependencies in the final image. We'll keep image size small (possibly using Python slim base or distroless) and utilize GCP's Artifact Registry scanning features for any new vulnerabilities. These practices, combined with our CI tests, ensure the Phase 2 deployment is robust and secure.

Milestone 6: Security Hardening and IAM Governance

Goal: Throughout Phase 2, we must incorporate security best practices, especially since we are dealing with live AI agents and cloud resources. This milestone highlights the completion of key security measures, many of which are implemented alongside earlier steps. The result is a system where each component has the minimum privileges it needs, and attack surface is minimized (important for a public-facing health-related site).

Implementation Tasks:
- **Least-Privilege for Pods:** For each deployed component (web app deployment, agent CronJob, any MCP tool service), use Kubernetes Service Accounts bound to GCP service accounts with tightly scoped roles. For example, the **FastAPI web app pod** can be restricted to only read from Firestore and invoke Vertex AI APIs – and nothing else [2]. The **content pipeline job** pod might have permission to write to Firestore (to save content) or commit to GitHub (if using a PAT, though we prefer using GitHub Actions for commits to avoid long-lived credentials in pods). No pod should have broad roles like Owner/Editor; each gets a custom role or predefined roles limited to its function. This way, even if a pod is compromised, it cannot affect other resources.
- **Network and Execution Security:** Ensure the GKE cluster uses best practices from Phase 1: private nodes or limited public exposure, and HTTPS only on the Ingress for the web app [32]. The MCP tool servers (if any run persistently) should be containerized and similarly locked down – for instance, a custom web-scraping tool might run as a microservice that only accepts queries via MCP and cannot access other internal services [33]. We also sandbox any untrusted execution: the LLM agents are constrained by the tools we give them, and MCP ensures they can't execute arbitrary code, only call the allowed functions [33].
- **Secrets Management:** No sensitive secret (API keys, etc.) is hard-coded. Use GCP Secret Manager or Kubernetes secrets for any credentials and mount/inject them at runtime to the pods that need them. Also prefer OAuth where possible (e.g. use Google's libraries to access Vertex AI with the service account token instead of API keys).
- **Audit and Logging:** Enable Cloud Logging and Monitoring for the cluster. The content pipeline's actions (especially the Publisher) should log what it published and when. Firestore writes create audit

logs as well. Set up alerting for any failures in the daily CronJob (e.g. if the job fails or doesn't produce content, send an alert). Also monitor costs for Vertex AI usage to catch any anomalies.
- **Finalize Documentation & Access:** Document the Phase 2 system, including how credentials and IAM are set up, for transparency. Limit developer access to production – since the site auto-updates, direct human interaction with production should be minimal. All changes (code or content) go through GitHub as intended. By locking down access and following the above security steps, we ensure the longevity coach site runs with minimal risk (no excessive permissions or exposed data) while still being dynamic and feature-rich.

Milestone 7: Review and Milestone Completion

(Checkpoint milestone tying everything together.) At this stage, Phase 2 delivers a working AI-driven website. We verify that:

- The **web UI** is live on GKE and shows a basic but functional site with longevity content.
- The **Firestore content system** is operational, storing articles/tips and feeding the frontend.
- The **multi-agent content pipeline** runs on schedule (or on-demand) and continuously generates new content that appears on the site.
- The **CI/CD pipeline** covers our development needs – any code updates or content additions go through automated tests and deployment.
- **Security and IAM** configurations are in place, with no component running with unnecessary privileges.

With Phase 2 completed, users can visit the site to read up-to-date AI-generated longevity articles in a friendly interface, satisfying the project's core objectives of dynamic content and robust architecture [34] [1]. The next phases would build on this by adding the interactive Q&A coach and further refining the system, but Phase 2 has established the crucial foundation of a self-updating longevity coach website.

Sources: The implementation plan is based on the project's design document and GCP best practices, including the use of Firestore for content storage [4], a FastAPI web framework [6], a multi-agent AI content creation pipeline using LangChain/LangGraph [10] [9] with tool integration via MCP [12], scheduled automation via Kubernetes CronJobs [25], and a CI/CD workflow with GitHub Actions for deploying code and content updates [20] [28]. These approaches adhere to the project's objectives and industry best practices to ensure a successful Phase 2 execution.

---

[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] Project_Plan_AI_Longevity_Coach_Website_on_GCP.pdf
file://file_0000000032e061f6a00667281638a0d4