The Agentic Development Framework: A Playbook for Building and Deploying "Retriever Study"

Section 1: Analysis of the Retriever Study Application

A comprehensive audit of the "Retriever Study" project reveals a well-conceived application with a strong architectural foundation, particularly in its backend security design. However, it also exhibits classic symptoms of uncoordinated parallel development, including discrepancies between planning documents and implemented code, and critical architectural gaps that must be addressed to achieve production readiness. This analysis establishes a definitive baseline, creating a single source of truth from which a structured, agent-driven development workflow can be successfully launched.

1.1. Architectural Blueprint: A Full-Stack Overview

The application's architecture is built on a modern, robust technology stack well-suited for its goals of creating an AI-powered, real-time collaborative platform. The separation of concerns between the frontend and backend is clear, and the choice of technologies reflects an understanding of scalable web application design.[1]

- **Frontend:** The user interface is a single-page application (SPA) built with React v18, leveraging React Router v6 for navigation. The project is bootstrapped with react-scripts (Create React App), providing a standardized build system.[1] State management for authentication is handled centrally through a React Context in AuthContext.js, a best practice for making user data available throughout the component tree.[1]
- **Backend:** The backend is a high-performance API constructed with FastAPI, a modern Python framework chosen for its asynchronous capabilities and automatic data validation powered by Pydantic.[1] This choice is ideal for an application that involves I/O-bound operations like database queries and external AI model calls.
- **Data Layer:** For local development, the application relies on a SQLite database (retriever_study_local.db), which is appropriate for its simplicity and ease of setup.[1] However, the project's dependencies and code structure clearly anticipate a migration

to a more robust, production-grade database. The inclusion of the asyncpg library in requirements.txt points to PostgreSQL as a planned target, while comments in the data access layer also mention DynamoDB as a potential alternative.[1]

- **AI/ML Services:** The core AI functionality is integrated directly into the FastAPI backend. It utilizes the sentence-transformers library for generating text embeddings for users and groups, and the transformers library for performing toxicity analysis on user-generated content and summarizing chat conversations.[1] This embedded approach simplifies the architecture for the MVP stage by avoiding the need for separate microservices.
- **Real-time Communication:** The application supports real-time chat functionality through a native WebSocket implementation within the FastAPI backend, managed in app/core/websocket.py.[1] The frontend connects to this service using a dedicated WebSocket client defined in frontend/src/services/socket.js, enabling bidirectional communication for an interactive chat experience.[1]

1.2. Backend Codebase Review: Production-Ready, with Caveats

The backend codebase demonstrates a sophisticated understanding of production security principles, yet it is in a transitional state regarding its data layer, creating a significant hurdle for deployment.

A primary strength of the backend is its multi-layered security architecture, meticulously documented in SECURITY_IMPLEMENTATION_GUIDE.md.[1] This "Security Onion" approach, implemented in app/core/security.py, shows a commitment to defense-in-depth that is rare in early-stage projects.[1] It includes distinct layers for rate limiting, request size validation, input sanitization, authentication, and security headers, each designed to mitigate specific attack vectors. This design indicates that security is not an afterthought but a core architectural consideration.

Despite this strength, the backend is hindered by an incomplete migration to a production-ready data layer. The codebase contains parallel data access modules: the synchronous local_db.py for SQLite and placeholder stubs in async_db.py for a future asynchronous system.[1] The application's main entrypoint, main.py, is littered with conditional logic such as if async_initialized and user_repo:, which switches between these two data access patterns.[1] While this facilitates local development, it introduces complexity and makes the code difficult to test and maintain. This transitional state represents a significant piece of technical debt that must be resolved before the application can be reliably deployed.

This technical state also highlights a deeper process issue: a significant disconnect between the project's documentation and its implementation. The FEATURES.md file, which should serve as a project tracker, lists core features like "Recommendations (cosine)" and "NL Search (vector)" as "TODO".[1] However, a direct review of the

main.py file reveals that the corresponding API endpoints (/recommendations and /search) are fully implemented, containing logic for generating embeddings and calculating cosine similarity.[1] Similarly, the frontend
GroupsList.js component contains the logic to call these endpoints.[1]
This discrepancy is a direct result of using multiple, uncoordinated AI agents for development. Without a central orchestrator enforcing a strict workflow where documentation is updated as a final step of any task, the project's state becomes fragmented. Different agents work from different assumptions, leading to wasted effort and the "half-finished work" described in the initial problem statement. The frontend_handover.md file is a manual, after-the-fact attempt to bridge this gap, but it is not a sustainable solution.[1] The fundamental problem is the lack of a synchronized, automated process for managing the development lifecycle. The framework proposed in this report is designed specifically to solve this by making task definition and status updates integral, non-negotiable parts of the workflow.

1.3. Frontend Codebase Review: A Solid but Insecure Foundation

The frontend application is well-structured, adhering to modern React conventions that promote maintainability and scalability. The use of a centralized service layer in services/api.js is a key strength, abstracting all backend communication into a single module.[1] This design makes it easy to manage API endpoints, handle authentication tokens, and standardize error handling across the entire application.[1] Likewise, the use of
AuthContext.js to manage authentication state provides a clean and efficient way to share user data and authentication status with any component that needs it, without resorting to prop-drilling.[1]
However, beneath this solid structure lies a critical architectural and security flaw in the authentication flow. As noted in the frontend_handover.md document, the application currently uses the JWT obtained directly from Google's OAuth response as the session token for authenticating with the backend API.[1] This is confirmed by the logic in
AuthContext.js and api.js, which store this token in localStorage and attach it to subsequent API requests.[1]
This implementation is problematic for several reasons:

1. **Incorrect Token Usage:** A Google ID token is intended to prove a user's identity to a server *once*. It is not designed to be a long-lived session token for an application's API.
2. **Lack of Control:** By using Google's token directly, the application's sessions are tied to Google's token expiration policies, which are short-lived and outside of the application's control. This prevents the implementation of a proper "remember me" functionality or a seamless token refresh mechanism.
3. **Bypassed Backend Logic:** This approach bypasses the backend's own session management and security logic. The backend should be the sole authority for issuing, validating, and revoking its own session tokens. This is essential for implementing features like server-side logout, tracking user sessions, and applying fine-grained

access control.

This authentication flow is the most significant piece of technical debt in the project. It must be addressed before any other feature work is undertaken, as it impacts nearly every authenticated user interaction. Correcting this will require a coordinated effort between backend and frontend development, making it the perfect inaugural task for the new agentic workflow.

1.4. Consolidated Project Roadmap

To eliminate the ambiguity caused by conflicting documentation and to provide a clear path to completing the MVP, the following consolidated roadmap has been created. It synthesizes the goals from FEATURES.md, PROJECT_GUIDE.md, and frontend_handover.md into a single, prioritized source of truth.[1] This roadmap will serve as the primary input for the Orchestrator when decomposing features into tasks for the AI agent team.

Feature ID	Feature Name	Domain	Priority	Current Status	Key Files & Endpoints
AUTH-01	Backend-Issued JWT Flow	Full-Stack	**CRITICAL**	Not Started	/auth/google/callback, api.js, AuthContext.js
BE-01	Complete Async DB Migration	Backend	High	Partially Implemented	main.py, data/async_db.py, data/local_db.py
FE-01	Full Profile Editing	Frontend	High	Partially Implemented	/users/me, Profile.js, api.js
FE-02	Chat Summarization Integration	Frontend	Medium	Not Started	/groups/{id}/messages/summary, GroupDetail.js
FE-03	Display Sender Name in Chat	Full-Stack	Medium	Not Started	/ws/groups/{id}, GroupDetail.js, websocket.py
FE-04	Group Detail Member List	Frontend	Medium	Not Started	GroupDetail.js
DEVOPS-01	Backend Lambda Packaging	DevOps	High	Not Started	requirements.txt, new build_lambda.sh script
DEVOPS-02	Frontend	DevOps	High	Not Started	frontend/build,

	S3/CloudFront Deploy				new deploy_frontend.sh script
QA-01	End-to-End User Flow Tests	QA	Medium	Not Started	backend/app/tests/

Section 2: The AI Subagent Development Framework

To address the challenges of duplicated work and fragmented progress, a new workflow is required. This Agentic Development Framework translates the abstract principles of parallelization and specialization from the Zach Wills article into a concrete operational model tailored for the "Retriever Study" project.[2] It establishes a system of specialized AI agents, managed by a human orchestrator, who work on well-defined, contextually isolated tasks.

2.1. Core Principles in Practice

The framework is built upon three core principles: Parallel Execution, Sequential Handoffs, and Context Isolation. Applying these principles systematically will transform the development process from chaotic to coordinated.

- **Parallel Execution for Speed:** This principle is about maximizing efficiency by executing independent tasks concurrently. Instead of a single agent attempting to modify the backend, then the frontend, then the tests, these tasks are assigned to specialist agents who work simultaneously. For example, when implementing feature FE-01 (Full Profile Editing) from the roadmap, the Orchestrator will dispatch two agents in parallel:
    - A Backend_Engineer_Agent will be tasked with modifying the PUT /users/me endpoint in main.py to accept and process the new courses and prefs fields.[1]
    - Simultaneously, a Frontend_Engineer_Agent will be tasked with updating the form in Profile.js to include input fields for these new properties and updating the updateUser function in api.js to send them.[1]

      The total time to get a working first draft of the full-stack feature is reduced to the time it takes to complete the longest of these two tasks, drastically accelerating development velocity.
- **Sequential Handoffs for Automation:** This principle creates an automated assembly line for quality assurance and process enforcement. The output of one agent becomes the input for the next, ensuring that every piece of code passes through a standardized

review process. Continuing the FE-01 example, once the Backend_Engineer_Agent has produced the updated Python code, its output (a complete file or a diff) is not immediately merged. Instead, it is automatically handed off to a Security_Auditor_Agent. This agent's task is to review the new code, using the stringent rules defined in SECURITY_IMPLEMENTATION_GUIDE.md as its primary context, to ensure that the new courses and prefs inputs are properly sanitized to prevent injection attacks.[1] Only after the
Security_Auditor_Agent approves the code can it proceed. This automates the code review process and enforces security best practices on every change.

- **Context Isolation for Quality:** This principle ensures high-quality output by providing each specialist agent with only the information it needs to perform its task, preventing context window limitations from degrading its performance. When working on FE-01, the Backend_Engineer_Agent receives a context that includes only the relevant backend files (main.py, core/security.py, data/local_db.py) and the security guide.[1] It is not burdened with the complexities of React state management or CSS styling. Conversely, the
Frontend_Engineer_Agent is given a context focused on Profile.js, api.js, and AuthContext.js, without needing to understand the underlying FastAPI database implementation.[1] This focused context allows each agent to apply its specialized knowledge with maximum effectiveness, resulting in higher-quality code for each component of the system.

2.2. Assembling Your AI Agent Team: Roles and Responsibilities

The successful implementation of this framework requires moving away from using general-purpose AI assistants and toward cultivating a team of specialized agents. Each agent is defined by a "master prompt" that assigns its persona, expertise, and constraints. The human developer acts as the **Orchestrator**, managing the workflow and making strategic decisions.

Agent Persona	Primary Tool(s)	Key Responsibilities	Core Expertise
Orchestrator	**You**	Decompose features, create UTDFs, manage handoffs, review final output.	Project Management, System Architecture
Backend_Engineer	Claude Code CLI / Gemini CLI	Write/modify FastAPI endpoints, data models, and database logic.	Python, FastAPI, Pydantic, SQL, Async
Frontend_Engineer	Cursor / Codex	Write/modify React components, hooks, and services.	React, JavaScript, CSS, State Management

QA_Engineer	Gemini CLI	Write unit, integration, and end-to-end tests.	Pytest, React Testing Library
Security_Auditor	Claude Code CLI	Review code for vulnerabilities based on project security guides.	Web Security, OWASP Top 10, FastAPI Security
DevOps_Engineer	Gemini CLI	Write deployment scripts, Dockerfiles, and CI/CD workflows.	AWS (Lambda, S3, CF), Docker, GitHub Actions

2.3. The Unified Task Definition File (UTDF): Your Single Source of Truth

The Unified Task Definition File (UTDF) is the central artifact of this workflow. It is a structured markdown file that replaces ad-hoc prompting with a formal, repeatable "ticket" system. For every unit of work, the Orchestrator will create a UTDF. This file serves as the complete, isolated context for an agent's task, ensuring absolute clarity on requirements, scope, and expected output. It is the primary tool for enforcing Context Isolation and is the key to eliminating ambiguity and duplicated work.
UTDF Template:

UTDF: -

1. Agent Assignment

- **Primary Agent:**
- **Reviewing Agent(s):**

2. Task Description

A clear, concise description of the goal. What should be achieved?
(e.g., "Modify the PUT /users/me endpoint to allow updating the user's courses and prefs fields.")

3. Acceptance Criteria

A bulleted list of conditions that must be met for the task to be considered complete.

- [ ] The endpoint must accept a list of strings for courses.
- [ ] The endpoint must accept a dictionary for prefs.
- [ ] Input must be sanitized before being saved to the database.
- [ ] A pytest unit test must be created to verify the update functionality.

4. Relevant Files & Context

A list of file paths and key documentation snippets the agent MUST use as context. This enforces Context Isolation.

- backend/app/main.py
- backend/app/data/local_db.py
- backend/SECURITY_IMPLEMENTATION_GUIDE.md (for sanitization rules)

5. Required Output Format

Strict instructions on how the agent should format its response.

- **Format:** Provide the complete, updated contents of each modified file.
- **Delimiters: Each file's content must be enclosed in a markdown code block with the language and file path specified, like so:python**

  File: backend/app/main.py

... file content...

Section 3: The Agentic Workflow: A Practical Playbook

This section provides the operational core of the framework, detailing the step-by-step procedures and providing the specific prompts necessary to direct the AI agent team effectively. This is the practical guide to executing the strategy defined in the previous section.

3.1. The Orchestrator's Protocol

As the Orchestrator, the developer's role shifts from writing every line of code to directing the AI agents who will generate it. This protocol outlines the systematic process to follow for every feature on the roadmap.

1. **Select a Task:** Begin by selecting the highest-priority incomplete feature from the **Consolidated MVP Feature Roadmap**. For example, start with AUTH-01: Backend-Issued JWT Flow.
2. **Decompose and Create UTDFs:** Break the feature down into its constituent parts. For AUTH-01, this requires both backend and frontend changes that can be developed in parallel. Create two separate UTDFs: AUTH-01-BE for the backend work and AUTH-01-FE for the frontend work. Fill out each UTDF with extreme precision, as this is the primary instruction set for the agents.
3. **Initiate the Primary Agents:** For each UTDF, combine the corresponding **Master Prompt** (from Section 3.2) with the full text of the UTDF. This combined text becomes the input for the designated AI agent and tool. For instance, the Backend_Engineer master prompt and the AUTH-01-BE UTDF are sent to the Claude Code CLI.
4. **Review and Refine:** The agent's initial output may require refinement. Review the generated code for correctness and adherence to project standards. If changes are needed, provide concise, iterative feedback. For example: "The logic is correct, but refactor the database call into a separate function in local_db.py and call it from main.py to maintain separation of concerns."
5. **Execute Sequential Handoffs:** Once the primary agent's output is satisfactory, initiate the review process. Take the code generated by the Backend_Engineer and create a new, smaller UTDF for the Security_Auditor. The task description would be: "Review the provided Python code for the new /auth/google/callback endpoint and ensure it complies with all principles in SECURITY_IMPLEMENTATION_GUIDE.md. Specifically, verify that the incoming Google token is validated correctly and that all database inputs are safe."
6. **Integrate and Commit:** After all primary and review tasks for a feature are complete and the generated code has been approved, integrate the changes into the main codebase. Finally, update the status of the feature on the project roadmap to "DONE."

3.2. Master Prompts for Agent Specialization

These master prompts are the configuration files for the AI agents. They establish the

persona, expertise, and operational constraints for each role. They should be prepended to every UTDF before being sent to the corresponding AI tool.

Backend_Engineer Master Prompt:

You are a senior backend engineer specializing in FastAPI and production-grade Python. Your name is Backend_Engineer. You adhere strictly to the provided project context and follow all security and style guidelines. You write asynchronous code where appropriate, use Pydantic for all data models, and ensure all database interactions are clean and efficient. You always provide complete, runnable code files as your output, formatted exactly as requested in the UTDF. You will now receive a Unified Task Definition File. Acknowledge your role and begin the task.

Frontend_Engineer Master Prompt:

You are a senior frontend engineer specializing in React and modern JavaScript. Your name is Frontend_Engineer. You build clean, maintainable, and responsive components. You manage state effectively using React Context and Hooks. You interact with backend APIs exclusively through the provided service layer (api.js). You follow the existing "Zara-inspired" design language and adhere strictly to the component structure and styling conventions in the provided files. You always provide complete, runnable code files as your output, formatted exactly as requested in the UTDF. You will now receive a Unified Task Definition File. Acknowledge your role and begin the task.

QA_Engineer Master Prompt:

You are a Quality Assurance Engineer specializing in automated testing for Python and React applications. Your name is QA_Engineer. Your sole focus is to write comprehensive tests that validate the acceptance criteria of a given task. For backend code, you write tests using the pytest framework. For frontend code, you use React Testing Library. Your tests are thorough, covering both success cases and edge cases. You always provide complete, runnable test files as your output, formatted exactly as requested in the UTDF. You will now receive a Unified Task Definition File. Acknowledge your role and begin the task.

Security_Auditor Master Prompt:

You are a cybersecurity specialist with expertise in web application security and the OWASP Top 10. Your name is Security_Auditor. Your only function is to review code for potential security vulnerabilities. You will be provided with code and relevant security documentation for the project. You must meticulously check for issues such as injection vulnerabilities, improper authentication, and data exposure. Your output should be a brief report identifying any vulnerabilities found and suggesting specific code changes for remediation. If no vulnerabilities are found, you will state "No vulnerabilities found." You will now receive a Unified Task Definition File. Acknowledge your role and begin the task.

DevOps_Engineer Master Prompt:

You are a DevOps Engineer with expertise in cloud-native applications, specifically on AWS. Your name is DevOps_Engineer. You specialize in creating automated, infrastructure-as-code solutions for deployment and CI/CD. You are proficient with Docker, AWS services (Lambda, API Gateway, S3, CloudFront), and GitHub Actions. You write clean, efficient, and well-documented scripts and configuration files. You will now receive a Unified Task Definition File. Acknowledge your role and begin the task.

3.3. End-to-End Walkthrough: Implementing Feature AUTH-01

This walkthrough demonstrates the entire framework in action by tackling the highest-priority task: fixing the authentication flow.
Step 1: Orchestrator Creates UTDFs (Parallel Execution)
The Orchestrator creates two UTDFs to be executed concurrently.
UTDF AUTH-01-BE:

UTDF: AUTH-01-BE - Implement Backend-Issued JWT Flow

1. Agent Assignment

- **Primary Agent:** Backend_Engineer
- **Reviewing Agent(s):** Security_Auditor, QA_Engineer

2. Task Description

The current system improperly uses a Google-issued token for session management. Create a new API endpoint that accepts a Google ID token, validates it, creates or updates a user in our database, and returns a secure, backend-issued JWT that will be used for all subsequent API requests.

3. Acceptance Criteria

- [ ] Create a new endpoint at POST /auth/google/callback.
- [ ] The endpoint should accept a JSON body with a single field: google_token.
- [ ] The endpoint must securely verify the google_token with Google's servers.
- [ ] Upon successful verification, it must use the user's Google ID (sub) to find an existing user or create a new one via the create_or_update_oauth_user function in local_db.py.
- [ ] A new, backend-signed JWT must be generated. This JWT's payload must contain the internal userId and the user's email.

- [ ] The endpoint must return a JSON response containing the access_token, refresh_token, and user profile information, matching the existing TokenResponse Pydantic model in main.py.

4. Relevant Files & Context

- backend/app/main.py
- backend/app/core/auth.py
- backend/app/data/local_db.py

5. Required Output Format

- **Format:** Provide the complete, updated contents of backend/app/main.py and backend/app/core/auth.py.
- **Delimiters:** Use markdown code blocks with file paths.

UTDF AUTH-01-FE:

UTDF: AUTH-01-FE - Integrate Backend-Issued JWT Flow

1. Agent Assignment

- **Primary Agent:** Frontend_Engineer
- **Reviewing Agent(s):** QA_Engineer

2. Task Description

Modify the frontend login process to exchange the Google token for a backend-issued JWT. The application must store and use this new backend token for all authenticated API calls.

3. Acceptance Criteria

- [ ] In frontend/src/services/api.js, rename the existing googleLogin function to exchangeGoogleToken.
- [ ] The exchangeGoogleToken function must now make a POST request to the /auth/google/callback endpoint, sending the Google credential.
- [ ] In frontend/src/pages/Login.js, update the handleLoginSuccess function to call the new exchangeGoogleToken service function.
- [ ] In frontend/src/context/AuthContext.js, the login function must now store the entire response object from the backend (which includes the access_token, refresh_token, and user object) in localStorage.
- [ ] The apiRequest function in api.js must be updated to read the access_token from this stored object.

4. Relevant Files & Context

- frontend/src/pages/Login.js
- frontend/src/services/api.js
- frontend/src/context/AuthContext.js

5. Required Output Format

- **Format:** Provide the complete, updated contents of all three modified JavaScript files.
- **Delimiters:** Use markdown code blocks with file paths.

Step 2: Sequential Handoff for Security Review
After the Backend_Engineer returns the modified Python files, the Orchestrator creates a new UTDF for the Security_Auditor.
UTDF AUTH-01-BE-REVIEW:

UTDF: AUTH-01-BE-REVIEW - Security Audit of New Auth Endpoint

1. Agent Assignment

- **Primary Agent:** Security_Auditor

2. Task Description

Review the provided Python code for the new POST /auth/google/callback endpoint. Verify that it securely handles the incoming Google token and creates a session.

3. Acceptance Criteria

- [ ] Confirm that the Google token is verified using Google's official libraries and is not trusted blindly.
- [ ] Ensure that no sensitive information is leaked in error messages.
- [ ] Confirm that the new backend-issued JWT is generated with a strong, secret key and does not contain sensitive user data in its payload beyond the userId and email.

4. Relevant Files & Context

- backend/SECURITY_IMPLEMENTATION_GUIDE.md

5. Required Output Format

- **Format:** A brief text report. If vulnerabilities are found, list them with recommended fixes. If not, state "No vulnerabilities found."

By following this structured, multi-step process, the critical authentication flaw is fixed in a parallelized yet secure manner. The final output is a full-stack implementation that has been reviewed for quality and security, all while minimizing development time.

Section 4: From MVP to Production on AWS

With the MVP features completed using the agentic workflow, the final goal is to deploy the application to AWS. This requires evolving the architecture from its development-centric state to a scalable, cloud-native model. The DevOps_Engineer agent will be instrumental in automating this process.

4.1. Architecting for the Cloud: Beyond SQLite

The current use of a single SQLite database file (retriever_study_local.db) is a major blocker

for production deployment.[1] In a serverless or multi-container environment, each instance of the application would be ephemeral and stateless, and a file-based database would lead to data loss and inconsistency. The application must be migrated to a managed, centralized database service.

The project's dependencies and existing code provide a clear path forward. The inclusion of asyncpg in requirements.txt indicates a preference for a PostgreSQL-compatible database.[1] This is an excellent choice, as it aligns with the structured, relational nature of the existing data models (
User, Group, Message).[1] While DynamoDB is mentioned as an alternative, migrating to a NoSQL model would require a significant and time-consuming refactoring of the data access layer.

Therefore, the recommended path is to use **AWS RDS for PostgreSQL (Serverless v2)**. This service provides the relational integrity the application is designed for, while its serverless nature is highly cost-effective for an application that may start with low or intermittent traffic. It can scale down to zero when not in use, which is ideal when operating on student credits, and scale up automatically as user load increases. The DevOps_Engineer agent can be tasked with provisioning this database using an infrastructure-as-code tool like AWS CloudFormation or Terraform.

4.2. The DevOps Agent Playbook

To automate the deployment process, the Orchestrator will direct the DevOps_Engineer agent to complete a series of tasks using UTDFs.

Task 1: Containerize the Backend Application

- **UTDF Goal:** Create a Dockerfile to package the FastAPI application into a standardized, portable container image.
- **Key Instructions for the Agent:**
    - Use a lightweight base image, such as python:3.11-slim, to keep the image size small.
    - Copy the requirements.txt file and install dependencies in a separate layer to leverage Docker's layer caching.
    - Copy the entire backend/ application code.
    - Expose the port that uvicorn will run on (e.g., port 80).
    - Set the CMD to launch the application using uvicorn app.main:app --host 0.0.0.0 --port 80.

Task 2: Deploy the Backend to AWS Lambda

- **UTDF Goal:** Create an AWS SAM template.yaml file to define the serverless backend infrastructure.
- **Key Instructions for the Agent:**
    - Define a resource of type AWS::Serverless::Function.
    - Set the PackageType to Image and reference the ECR repository where the

Docker image from Task 1 will be stored.
    - Define an API Gateway trigger for the Lambda function, configured to proxy all requests (/{proxy+}) to the FastAPI application.
    - Define environment variables for the Lambda function. Crucially, sensitive values like the DATABASE_URL and JWT_SECRET_KEY should not be hardcoded. Instead, instruct the agent to configure the template to fetch these values from AWS Secrets Manager at runtime.

Task 3: Deploy the Frontend to S3 and CloudFront

- **UTDF Goal:** Write a shell script (deploy_frontend.sh) to build and deploy the React application.
- **Key Instructions for the Agent:**
    - The script must first run npm run build inside the frontend/ directory.
    - It must then use the AWS CLI command aws s3 sync to upload the contents of the frontend/build directory to a specified S3 bucket.
    - The script should configure the S3 bucket for static website hosting.
    - It should then create or update an AWS CloudFront distribution that uses the S3 bucket as its origin. The agent must be instructed to configure the CloudFront distribution to handle SPA routing by creating a custom error response that redirects all 404 Not Found errors back to /index.html with a 200 OK status.

Task 4: Create a CI/CD Pipeline

- **UTDF Goal:** Create a GitHub Actions workflow file (.github/workflows/deploy.yml) to automate the entire deployment process.
- **Key Instructions for the Agent:**
    - The workflow should be triggered on every push to the main branch.
    - It should define two separate jobs, deploy-backend and deploy-frontend, that can run in parallel.
    - The deploy-backend job will check out the code, build the Docker image, push it to Amazon ECR, and then deploy the SAM application.
    - The deploy-frontend job will check out the code, set up Node.js, install dependencies, and then execute the deploy_frontend.sh script.
    - The agent must be reminded to use GitHub Secrets to store AWS credentials (AWS_ACCESS_KEY_ID and AWS_SECRET_ACCESS_KEY) securely.

Section 5: Conclusion: A Blueprint for Future Projects

The "Retriever Study" project, when completed through this framework, will represent more than just a functional, AI-powered web application. It will serve as a powerful testament to a sophisticated and modern approach to software development. The challenges initially faced—duplicated work, fragmented progress, and inconsistent quality—are not unique; they are common pitfalls in any complex project, especially those leveraging multiple, powerful development tools.

By implementing the Agentic Development Framework, the role of the developer is elevated from a mere coder to an **Orchestrator**. This framework imposes a structure that channels the capabilities of specialized AI agents into a coordinated, efficient, and quality-driven process. The use of a single source of truth in the **Consolidated Project Roadmap** eliminates ambiguity, while the **Unified Task Definition File (UTDF)** ensures that every unit of work is clearly defined, contextually isolated, and testable. The principles of **Parallel Execution** and **Sequential Handoffs** transform the development lifecycle into an automated assembly line, accelerating progress while simultaneously enforcing rigorous quality and security checks. The final deployed application on AWS will be a significant technical achievement. However, the true value of this endeavor lies in the process itself. The ability to articulate and demonstrate this systematic approach to managing a team of AI subagents to build, test, and deploy a production-grade application is a skill set that is at the forefront of the software engineering industry. This project, therefore, becomes not just an entry on a resume, but a compelling narrative about professional-grade project management, architectural foresight, and the strategic application of cutting-edge technology—a combination that will be a profound differentiator in the pursuit of new-grad software engineering roles.

Works cited

1. rodriguezzfabricio/retriever-study
2. How to Use Claude Code Subagents to Parallelize Development ..., accessed September 20, 2025, https://zachwills.net/how-to-use-claude-code-subagents-to-parallelize-development/