ChatGPT

BrainStackStudio System Blueprint

Executive Summary

BrainStackStudio is building an **AI-native automation platform** to streamline business operations across multiple product lines (e.g. *MyRoofGenius* for roofing estimates, *TemplateForge* for content templates). The target solution is a **persistent, modular backend** powered by FastAPI and multiple AI agents (Claude, GPT-4 Codex, Google Gemini, etc.), combined with responsive frontends for both operators and end-users. The platform orchestrates complex workflows – from generating documents and code to updating project boards – using large language models (LLMs) as the "brain" for decision-making and content creation. All actions and outputs are stored as *persistent memory* (via Supabase) for learning and retrieval, ensuring that the system becomes smarter and more context-aware over time [1].

In the target state, BrainStackStudio's automation system acts as an **AI co-pilot** for the business: it can generate standard operating procedures, write and deploy code for new features, respond to user inquiries with up-to-date information, and coordinate tasks on external services. Each brand (MyRoofGenius, TemplateForge, etc.) will leverage the same core backend with brand-specific configurations and prompt templates, enabling reuse of automation capabilities across domains. The architecture emphasizes **modularity** (clear separation of concerns for routes, agents, memory, tasks, and UI), **scalability** (containerized deployment, task queues, and vector databases), and **retrieval augmented generation (RAG)** (integrating the company's knowledge base into AI prompts for accurate, context-rich outputs). By combining multiple specialized AI agents in a controlled workflow, the system can handle end-to-end processes with minimal human intervention while still allowing human oversight through an operator dashboard. In summary, this blueprint outlines a full-stack solution where **Claude** excels at planning and documentation, **Codex/GPT-4** at code generation, **Gemini** at content optimization, and **Perplexity** at research – all coordinated through a unified FastAPI backend that logs every step for continuous learning.

Repository & Architecture Audit

Current State: The existing repository (`mwwoodworth/fastapi-operator-env`) provides a foundation: a FastAPI server (`main.py`) with endpoints for running tasks, a lightweight task execution framework, and a Next.js dashboard for monitoring [2] [3]. Tasks are defined in a `codex/tasks/` module and executed via a central dispatcher (`brainops_operator.py`) that registers each task by scanning the folder [4] [5]. Completed tasks and outputs are logged to both a Supabase database (for persistent memory) and local files [6] [7]. The system integrates with external services through webhook endpoints (e.g. Slack commands, Stripe events, Make.com triggers) [8] and uses Celery for asynchronous job queuing (returning a `task_id` and allowing status checks) [9] [10]. A Next.js front-end (located in `dashboard_ui/`, with static export served at `/dashboard/ui`) provides real-time charts and a basic chat interface [11]. This dashboard shows metrics like task throughput and allows simple interactions (e.g. a Slack-like chat to trigger tasks). The groundwork for **RAG** is present: a Supabase-backed `memory` table stores conversation history and task outputs, and vector search is enabled via a Postgres function (`match_documents`) for

semantic lookup [12] [13]. Additionally, environment-configured API keys enable connections to LLMs (Claude, OpenAI, Gemini) and services (Slack, ClickUp, Notion, etc.) [14] [15].

Gaps and Improvement Areas: The current codebase, while functional, has some architectural gaps that this blueprint addresses:

- *Separation of Concerns:* Many API routes are defined in the monolithic `main.py`, and the `codex` module mixes various domains (AI calls, tasks, integrations, memory) under one namespace. We will refactor into a clearer structure with dedicated sub-packages for **routes**, **tasks**, **agents**, **memory**, and **webhooks** for better maintainability.

- *Multi-Agent Orchestration:* The system currently routes prompts to models in code (via simple logic in `utils/ai_router.py`) – e.g. tasks containing `"code"` use Gemini vs. `"summary"` use Claude [16]. However, complex multi-step workflows (Claude → Codex → Gemini) are not declaratively defined. We introduce a **LangGraph** configuration to explicitly map out agent workflows as a directed graph, instead of hard-coding sequences. This will make it easier to add new pipelines and adjust routing logic without modifying code.

- *Folder Structure:* The repository needs to be organized as a **monorepo** supporting both backend and frontend, following Turborepo conventions. Currently, the Next.js dashboard lives in `dashboard_ui` and some UI code is in `frontend/`, which is confusing. We propose a consolidated structure (see below) that clearly delineates the front-end app(s) and back-end app, as well as shared libraries (e.g. an SDK for API calls).

- *Extensibility:* Additional integration points (e.g. Notion sync, advanced calendar or Kanban views from the backlog [17]) are only partially implemented or still ideas. The new architecture will allocate proper modules/placeholders for these (e.g. a `integrations/notion.py` route, a scheduler for recurring tasks) to facilitate development.

- *Testing & CI:* We will strengthen the continuous integration pipeline to cover the new multi-agent flows and front-end build, ensuring that Claude-generated docs and Codex-generated code can be safely verified (linting, unit tests) before deployment.

Proposed Monorepo Structure: Below is an updated folder layout for the project, grouping related components and following a monorepo approach using Turborepo (for managing builds and dependencies across multiple apps). The backend and frontend will reside in a single repository, enabling atomic changes across the stack and easier sharing of model schemas and types.

```
brainstackstudio/
├── apps/
│   ├── backend/            # FastAPI application (BrainOps Operator Service)
│   │   ├── main.py         # Application entry (mounts routers, startup
events, etc.)
│   │   ├── routes/         # FastAPI route modules (each corresponds to a
feature area)
│   │   │   ├── tasks.py        # Endpoints for task operations (run, status,
```

```
design, etc.)
│   │   │   ├── auth.py          # Authentication endpoints (token, refresh,
logout)
│   │   │   ├── memory.py        # Endpoints for memory RAG (query, write,
update)
│   │   │   ├── webhooks.py      # Endpoints for external webhooks (Slack,
ClickUp, Stripe, etc.)
│   │   │   └── agents.py        # Endpoints for agent coordination (inbox
approvals, pipelines)
│   │   ├── agents/          # Multi-agent system logic
│   │   │   ├── langgraph.yaml   # YAML definition of agent graph (nodes, edges,
routing rules)
│   │   │   ├── base.py          # Core classes for agent nodes, graph
execution, context management
│   │   │   ├── claude_agent.py  # Wrapper for calling Claude (Anthropic) with
standardized interface
│   │   │   ├── codex_agent.py   # Wrapper for calling GPT-4 (OpenAI Codex) for
code generation
│   │   │   ├── gemini_agent.py  # Wrapper for calling Google Gemini for
content/SEO tasks
│   │   │   └── search_agent.py  # Wrapper for calling Perplexity or web search
for research
│   │   ├── tasks/           # Task implementations (business logic, uses
agents)
│   │   │   ├── __init__.py      # Task registry initialization (auto-register
tasks)
│   │   │   ├── autopublish_content.py   # Example task: publish article (calls
Claude + integration)
│   │   │   ├── generate_product_docs.py # Example task: generate product docs
via Claude
│   │   │   ├── generate_roof_estimate.py# Example task: roof estimate
calculation (mix of AI + logic)
│   │   │   ├── ... other task files ... # (each defines TASK_ID, run() and
optional stream() method)
│   │   │   └── prompt_templates/        # Prompt text templates for tasks (see
Prompt Architecture)
│   │   │       ├── claude/ ... (.md)    # Claude prompt templates (SOP,
blueprint, etc.)
│   │   │       ├── codex/ ... (.md)     # Codex prompt templates (code
generation instructions)
│   │   │       ├── gemini/ ... (.md)    # Gemini prompt templates (SEO
optimization, summaries)
│   │   │       └── research/ ... (.md)  # Templates for injecting search
results into prompts
│   │   ├── memory/          # Memory and knowledge management
│   │   │   ├── models.py        # Pydantic models and schema for memory
records, docs, sessions
```

```
│   │   │   ├── memory_store.py  # Functions to save/query memory entries in
Supabase [18] [19]
│   │   │   ├── knowledge.py     # Functions for document chunking, embedding,
and retrieval
│   │   │   ├── supabase_client.py# Supabase connection initialization (pgvector
setup, etc.)
│   │   │   └── vector_utils.py  # Helpers for embedding generation (calls
OpenAI embeddings)
│   │   ├── integrations/    # External service integrations and webhook
handlers
│   │   │   ├── slack.py         # Slack slash-command and event handling
(approval, queries)
│   │   │   ├── clickup.py       # ClickUp webhook and API client (for task
syncing)
│   │   │   ├── notion.py        # Notion API integration (import/export tasks,
if applicable)
│   │   │   ├── make.py          # Make.com webhook handler (trigger tasks via
secret)
│   │   │   └── stripe.py        # Stripe webhook handler (e.g. on new sale ->
trigger onboarding)
│   │   ├── core/             # Core configuration and app setup
│   │   │   ├── settings.py      # Pydantic BaseSettings for env variables (API
keys, URLs) [20] [15]
│   │   │   ├── scheduler.py     # Background scheduler (for recurring tasks,
delayed tasks)
│   │   │   ├── security.py      # Auth/security utilities (password hashing,
JWT handling)
│   │   │   └── logging.py       # Logging configuration (JSON logging, Slack
alerts, Supabase sinks)
│   │   ├── db/               # Database (Supabase/Postgres) related files
│   │   │   ├── schema.sql       # SQL schema or Alembic migrations for tables
(tasks, memory, etc.)
│   │   │   └── migrations/      # Alembic migration scripts
│   │   ├── tests/            # Backend tests (unit and integration)
│   │   └── Dockerfile        # Docker image for FastAPI app
│   └── frontend/            # Next.js application(s) for UI
│       ├── brainops-admin/    # Admin Operator Dashboard (BrainStackStudio)
│       │   ├── pages/             # Next.js pages (or app/ directory if using
Next 13+)
│       │   │   ├── index.tsx         # Marketing site homepage (BrainStack
Studio public info)
│       │   │   ├── dashboard/[…].tsx # Dashboard UI pages (protected routes for
operator, PWA capable)
│       │   │   └── api/contact.ts    # Example API route (for contact form
forwarding to backend)
│       │   ├── components/       # Shared React components for the dashboard
│       │   │   ├── TaskList.tsx      # Admin task inbox list & approval buttons
```

```
│       │   │   ├── PromptPanel.tsx   # Panel to submit AI tasks (choose model,
input prompt)
│       │   │   ├── OutputStream.tsx  # Streaming output viewer (Server-Sent
Events handling)
│       │   │   ├── MarkdownViewer.tsx# Render markdown with formatting (for
docs, SOPs)
│       │   │   └── Charts.tsx        # Live charts for task metrics (through
websockets or polling)
│       │   ├── utils/            # Frontend utilities
│       │   │   ├── apiClient.ts      # Helper to call backend API (fetch with
auth, etc.)
│       │   │   ├── auth.ts           # Auth token storage & management (JWT in
localStorage, CSRF)
│       │   │   └── …
│       │   ├── public/          # Static assets (icons, manifest.json for PWA)
│       │   └── package.json     # Dependencies for admin dashboard
│       └── myroofgenius-copilot/ # User-facing Copilot UI for MyRoofGenius
(similar structure to admin)
│           ├── pages/…           # Pages for user interactions (e.g. chat
interface, report viewer)
│           ├── components/…      # Possibly a subset of components
(PromptPanel, OutputStream reused)
│           └── package.json      # Dependencies for user app (could be combined
with admin as one app)
├── packages/                # (Optional) Shared packages for Turborepo
│   ├── sdk/                   # TypeScript SDK for the backend API
│   │   ├── index.ts              # Exports API client, types for tasks,
memory, etc.
│   │   └── services/DefaultService.ts # Example service wrapper for default API
calls
│   └── ui-components/         # Shared React UI components (if any shared
across admin/user apps)
├── docs/                     # Documentation (markdown files for SOPs,
blueprints, etc.)
│   ├── README.md                # High-level README for repository
│   ├── architecture.md          # (This document or an evolved form of it)
│   ├── prompts/…                # Documentation on prompt templates and usage
guidelines
│   └── production_checklist.md  # Deployment checklist 21 22
├── turborepo.json           # Turborepo configuration (tasks pipeline for
build, lint, dev, etc.)
└── .github/
    ├── workflows/ci-cd.yml      # GitHub Actions for CI tests and CD deployment
    └── issue_templates/…        # Issue/PR templates (if needed)
```

Highlights of the New Structure: We separate the FastAPI **routes** by domain (`tasks.py`, `memory.py`, `webhooks.py`, etc.), which will make the API easier to navigate and maintain. The **agents** directory contains the logic for each AI agent and the orchestrator (`langgraph.yaml` plus helper code) – this cleanly encapsulates multi-agent workflows away from business logic. The **tasks** directory holds individual automation tasks (each as a self-contained module with `run()` function, analogous to the current `codex/tasks/*` files). Prompt templates are stored under `tasks/prompt_templates/` organized by use-case, making it easy to edit prompts without touching code. The **memory** package manages all interactions with the Supabase database (storing and querying memory and documents) and vector embedding operations. External service hooks (Slack, ClickUp, etc.) live under **integrations**, decoupling them from core logic. The frontend is split into potentially two Next.js apps: one for the BrainStackStudio operator admin dashboard and one for the MyRoofGenius end-user UI (they could alternatively be a single Next.js project with runtime switches, but separate apps allow independent deployment and branding). Both apps share common components and API client logic, which can be abstracted in a shared `packages/sdk` or `packages/ui-components` if needed.

We recommend using **Docker** for deployment consistency: one image for the FastAPI backend (with Uvicorn, Celery, etc.) and separate build output for the Next.js frontend (exported as static or served via Vercel). The monorepo with Turborepo will allow parallel building of backend and frontend, caching, and easier dependency management. In production, the FastAPI app will serve the static dashboard for the admin UI at `/dashboard/ui` (as currently done) [3], and the user-facing frontends can be deployed to Vercel or a separate static host (or even served similarly if desired). This structure sets the stage for a scalable, multi-application ecosystem that still shares one source of truth for tasks and memory.

Automation Pipelines

A core feature of BrainStackStudio's platform is the ability to run multi-step **automation pipelines** that involve different AI agents and tools. Below we define the key automation pipelines the system will support, replacing what used to be manual processes or external scripts (like Make.com scenarios) with AI-driven workflows. Each pipeline is orchestrated via the LangGraph multi-agent system (defined in the next section), ensuring a clear and flexible execution order. Key pipelines include:

- **Claude Markdown Doc Generation:** *Use case:* Generate structured documents (e.g. SOPs, blueprints, changelogs, content drafts) in Markdown format using Anthropic Claude. *Pipeline:* The trigger could be a Slack command or an HTTP request carrying parameters (e.g. "create SOP for onboarding process"). The system routes this request to **Claude** (with the appropriate prompt template selected for the document type). Claude produces a well-structured Markdown document, following our templates for headings, formatting, and embedded metadata. This Markdown might represent a Standard Operating Procedure, a weekly report, or a design blueprint. The result is stored in memory and optionally in the knowledge base (as a doc entry), and can be sent onward (e.g. posted to a Notion page or saved as a PDF). This pipeline **replaces manual doc writing or Make.com templating** with an AI that writes the initial draft, which can then be reviewed by a human. All generated docs will adhere to consistent formatting rules (like including titles, dates, and appropriate section headings) so they are immediately usable.

- **Claude → Codex Dev Handoff (Spec to Code):** *Use case:* Accelerate development by having AI write code from high-level specs. In this pipeline, **Claude** first produces a detailed **technical specification**

or code blueprint in Markdown (for example, a description of new API endpoints, data models, and even pseudo-code for functions). This spec will include code sections or structured instructions that a coding agent can follow exactly. The pipeline then feeds Claude's output to **Codex (GPT-4)**, which acts as the code generator. Codex reads the Markdown spec and writes actual source code files (Python, TypeScript, etc.) as outlined. We enforce a strict format for this handoff – e.g. the Claude-generated markdown uses fenced code blocks annotated with file names and content, so that Codex can unambiguously create each file. For example, Claude's doc might contain:

````
**File: tasks/forecast_agent.py**
```python
# Code for weekly forecast agent
import datetime
...
````

```

Codex will parse this and produce `forecast_agent.py` with the given content. The system can automate this flow via a special endpoint or CLI command: the developer (or an automated trigger) provides a request for a new feature, Claude's *Blueprint* is created, then Codex writes the code. The new code can be automatically inserted into the repository or staged for review. This pipeline turns natural language design into working code, dramatically reducing implementation time. We replace the old Makefile or manual coding steps with an AI pair-programming loop. A **safety check** is included: after Codex generates code, the CI pipeline runs tests and linting to ensure nothing is broken (see CI/CD section). Only if tests pass will the changes be deployed, closing the loop from specification to production.

- **Perplexity Research Inject → Claude Rewrite:** *Use case:* Answer questions or create content with up-to-date external information. *Pipeline:* When a task requires knowledge beyond the internal memory (e.g. "Write a market analysis of solar panel adoption in 2025"), the system first invokes a **Research agent** (using Perplexity or a similar search engine API) to gather relevant facts, figures, or references from the web. For example, a `research_agent` node takes the user's query and returns a summary or a set of key points with citations. This research result is then **injected into a Claude prompt** – we have a template that might say: *"Using the following research findings, write a comprehensive answer/report: [insert findings]."* Claude then produces a final document or answer that is grounded in the retrieved information. The benefit of this pipeline is that Claude's output will be both up-to-date and cite its sources (we can prompt it to include the references provided). This automation replaces having a human do manual Google searches and then asking AI – instead it's one seamless loop. The Perplexity agent's output can also be stored as a special memory record (tagged as `research`) so that the context is logged for future reference or audits. This pipeline is crucial for knowledge work like generating blog posts, reports, or answers that require real-time or factual accuracy.

- **Gemini SEO Optimizer Loop:** *Use case:* Refine and enhance content for SEO or other quality metrics using iterative AI feedback. *Pipeline:* After content is generated (by Claude or a human), we use **Google Gemini** (an LLM from Google's PaLM API) to analyze the content for improvements. For instance, if Claude drafts a blog article, the Gemini agent can take that draft and provide an "SEO optimization" pass – suggesting keyword additions, clearer headings, meta descriptions, or

restructuring for better engagement. In practice, the pipeline might: (1) call Gemini with a prompt like "Here is an article draft and its target audience/keywords – suggest improvements or rewrite sections where necessary to maximize SEO." (2) Gemini returns an improved version or a list of suggestions. We could either have Gemini directly rewrite the content (single-step) or enter a loop where Claude and Gemini refine in turn: e.g. Gemini flags issues, Claude (or GPT-4) fixes them, and repeat if needed. This *feedback loop* continues until a certain quality threshold is met or a set number of iterations is done. The final optimized content is then ready for publishing. Automating the SEO polishing step ensures that content produced by the platform is high-quality and saves human editors time. This pipeline leverages Gemini's strengths (which might include nuanced understanding of language and Google's ranking criteria) to complement Claude's initial creative generation, resulting in a more effective output.

- **File Output & Publishing Workflows (SOPs, Reports, Bundles):** *Use case:* Take final AI outputs and distribute or archive them appropriately. *Pipeline:* Many tasks culminate in producing files – e.g. a PDF report for a client, a bundle of markdown files for documentation, or a CSV of processed data. The system automates these post-processing steps. For example, an **SOP generation** task after producing markdown might trigger a conversion to PDF (using a service or library) and then upload that PDF to a designated storage or email it to stakeholders. A **reporting workflow** (like weekly metrics report) might compile multiple AI outputs: perhaps run several sub-tasks (data fetch, summary generation, chart image generation) and then bundle these into one package (markdown + images zipped or a nicely formatted PDF). *Bundles* refer to grouping outputs – e.g. a "launch bundle" could include a blog post, a newsletter draft, and social media posts, all generated by respective tasks and then packaged together for review. The pipeline coordinating this would use the agent graph to run each required generation in sequence and then a final step to collect the results. For instance, Claude could produce a blog article and a Twitter thread (different templates) in parallel, then a final agent combines them into a single deliverable (like a ZIP or a Notion update). The platform's role is to ensure each file is named, formatted, and stored properly – using internal APIs like `/knowledge/doc/upload` to save in the knowledge base or sending via webhook to external systems (e.g. uploading to a CMS via Make.com). Essentially, the system is an end-to-end publishing pipeline: **generate content -> optimize it -> format it -> deliver it**, replacing what might previously have been many manual steps across different tools.

Each of these pipelines is defined in the **LangGraph** (see below) so that the sequence of agents and actions is declarative. Operators can trigger these pipelines manually (via the dashboard UI or Slack commands) or they can be triggered automatically by events (e.g. a Stripe sale triggers the *sync_sale* task, which might run a bundle pipeline to onboard a customer). All pipelines are designed to be **retryable** and **traceable**: if any step fails, the error is logged and a retry or human review can be initiated (the Task Engine will handle retries and escalation as described later). By using LLMs as the building blocks, these automation pipelines remain flexible – we can easily change a prompt to adjust the outcome or insert an extra verification step (for example, have GPT-4 double-check Gemini's SEO suggestions, etc.) without rewriting complex code. This is a key advantage of an AI-native architecture for automation.

## LangGraph Multi-Agent Routing

To coordinate the above pipelines, we introduce **LangGraph**, a YAML-defined graph of agents and decision nodes that routes tasks to the appropriate AI model or tool in sequence. LangGraph provides a high-level *workflow schema* for multi-agent interactions, making the orchestration **declarative**. Rather than

hardcoding chains of calls, we define a graph where each **node** represents an action (an AI agent invocation or a function) and **edges** dictate the flow from one node to the next.

**YAML Structure:** The `agents/langgraph.yaml` file will define the nodes, edges, and any conditional routing. Below is an illustrative snippet of how this could look:

```
agents:
  - id: claude_max
    type: llm
    model: claude-2-100k
    role: "DocumentationGenerator"
    prompt_template: "claude/blueprint.md"   # reference to prompt template
    input_keys: ["specification"]   # expects a 'specification' text
input
    output_keys: ["blueprint_markdown"]   # produces a markdown doc
  - id: gpt4_codex
    type: llm
    model: gpt-4-code
    role: "CodeWriter"
    prompt_template: "codex/implement_code.md"
    input_keys: ["blueprint_markdown"]
    output_keys: ["code_files"]   # could be a list of files with
content
  - id: gemini
    type: llm
    model: gemini-2.5-pro
    role: "ContentOptimizer"
    prompt_template: "gemini/seo_optimize.md"
    input_keys: ["draft_content"]
    output_keys: ["optimized_content"]
  - id: perplexity
    type: tool
    tool: "web_search"   # denotes an external search tool
usage
    input_keys: ["query"]
    output_keys: ["research_summary", "sources"]
  - id: combiner
    type: function
    function: "combine_outputs"   # a custom function to merge results
    input_keys: ["part1", "part2"]
    output_keys: ["combined"]
flows:
  - name: "spec_to_code"   # pipeline definition
    start: claude_max
    steps:
      claude_max:
        next: gpt4_codex   # after Claude produces spec, go to
```

```
Codex
      gpt4_codex:
        next: null                          # end of pipeline (code files ready)
  - name: "research_and_answer"
    start: perplexity
    steps:
      perplexity:
        next: claude_max
      claude_max:
        params:                             # override prompt for answer
generation
          prompt_template: "claude/research_answer.md"
        next: null
  - name: "seo_loop"
    start: gemini
    steps:
      gemini:
        next: claude_max
      claude_max:
        params:
          prompt_template: "claude/revise_with_feedback.md"
        next: gemini                        # loop back to gemini for another
evaluation
      gemini:
        condition: "${iterations} < 2 and needs_improvement"  # pseudo-
conditional
        next: claude_max
```

(Note: The above is a conceptual example to illustrate structure. Actual implementation may differ in syntax.)

In this YAML, `agents` defines each node: **Claude** (id `claude_max`) is configured as a large-context model for documentation, **GPT-4 Codex** for coding, **Gemini** for content refinement, **Perplexity** as a search tool, and a custom combiner function. Each agent node has specified **input keys** and **output keys**, which describe what data it expects and produces. This ensures that the output of one node can be passed as input to the next. The `flows` section then strings these nodes together for specific named pipelines. For example, the `spec_to_code` flow routes from Claude to Codex, whereas `research_and_answer` first does a web search via Perplexity, then passes the results to Claude (with a special prompt template tuned for Q&A using provided sources). The `seo_loop` demonstrates how we can even encode a loop: after Gemini provides feedback, Claude revises, and if conditions meet (like if the content still needs improvement and we haven't looped too many times), it goes back to Gemini. This conditional or iterative logic can be supported either by YAML (with a `condition` field as shown) or handled in code by the orchestrator reading the graph definition.

Agent Schemas & Roles: Each agent node can have a **schema** for its inputs/outputs. For LLMs, the input is typically a text prompt (constructed from one or more context pieces) and the output is typically text (which might represent code, or a document, or a list, etc.). We define agents' roles to clarify their responsibility in the workflow. For instance, `DocumentationGenerator` Claude will always output a full Markdown

document given some spec context, whereas `CodeWriter` GPT-4 will output structured code (perhaps in JSON like `{"filename": "content"}` pairs for multiple files, or just text with separators). Defining these roles and formats explicitly means each agent's output can be programmatically parsed and fed forward. We will maintain a mapping of agent `id` to a small Python class or function that knows how to call that agent: e.g. `claude_agent.py` will have something like `def run(prompt: str) -> str` (and possibly a streaming version), similarly `codex_agent.py` and `gemini_agent.py` will call their respective APIs (Anthropic, OpenAI, Google) with the given prompt and return the result.

Routing Logic: The LangGraph orchestrator (`agents/base.py` or similar) will read the YAML graph and handle the execution logic. When a request comes in to run a pipeline (for example, the `/task/generate` endpoint might correspond to a flow name in the YAML), the orchestrator starts at the `start` node and calls the associated agent or function. Each node's output is stored in a context dictionary, and then passed to the next node as per the `next` pointer. The orchestrator can also evaluate conditions for branching: e.g. a decision node could examine the content or some flag to choose a different next node (we can extend YAML to allow a node to have multiple `next` options with conditions, or have a special "router" node type). For instance, we might have a classifier agent at the start that decides which pipeline to follow ("chain selection"); however, in our design we usually know which flow we want from the start, so that might not be needed.

One important aspect is **memory context integration**: before an agent prompt is constructed, we often need to inject relevant memory or knowledge. Rather than clutter the YAML with that, the orchestrator could automatically handle it based on agent type or a flag. For example, if an agent node has `use_memory: true`, the orchestrator will fetch recent memory or do a vector search (if a `query` key is present) and append the results to the prompt (likely via a template). This way, something like the chat agent or Claude answering a question will always include a "Recent memory" section or relevant docs section in its prompt [23] [24]. In practice, we have endpoints like `/chat` already doing this memory injection [25] – those will be refactored to use the unified LangGraph approach, so that if we ever change how we retrieve memory (say, increase the number of past messages or include semantic matches), we do it consistently for all agents that need it.

FastAPI Integration: Each flow can be exposed via the API for ease of use. For example, we might have `POST /pipeline/run` that accepts a JSON like `{"flow": "spec_to_code", "inputs": { ... } }` to trigger a specific flow by name. However, more friendly is to have dedicated endpoints for common use cases: e.g. `/task/dev-handoff` could internally call the `spec_to_code` flow, `/task/answer` calls the `research_and_answer` flow, etc. The *agents.py* route can define these and simply delegate to a LangGraph executor. This keeps the external API simple while the internal logic is governed by the YAML. Furthermore, we can have a **routing logic auto-mode**: the `AI_ROUTER_MODE` in settings was previously "auto/claude/gemini" to choose a model [26]. In the new design, if mode is "auto", we could dynamically pick a flow or agent based on input. For instance, a general `/chat` endpoint could check the query: if it contains a programming-related request, route to a *"coding assistant"* flow (maybe GPT-4 only); if it's a planning request, route to Claude, etc. This essentially extends the current simple router to a more complex decision graph – potentially a classifier LLM could be used to decide (we could have a node that outputs a route). But to keep things deterministic, initial routing can be done with rules/keywords (like before) or via specifying which UI button was clicked (e.g. user explicitly chooses "Ask Brain (Claude)" vs "Ask Data (Gemini)" in the UI). The YAML will allow defining composite flows too – e.g. a `chain` that actually runs multiple agents concurrently or sequentially (like get both Claude and GPT-4 answers then combine). The orchestrator needs to support parallel branches (maybe executed as async tasks) if we want to do things

like get answers from two models at once. That can be an extension where a node can have multiple `next` targets without waiting, and a subsequent node that gathers results (the `combiner` function node in the example).

In summary, LangGraph gives us a **flexible, configurable brain** for the system. We can update `langgraph.yaml` to tweak our automation pipelines (add a new step, change which model is used for a role, etc.) without altering the Python code – the orchestrator will adapt. It also provides a clear map for debugging: an operator can see a visualization of a flow (we could even build a UI component in the dashboard to display the graph of a run) and pinpoint where things went wrong or which agent produced what output. Each agent's outputs will be logged with a `node_id` in the memory, so we preserve the trace of multi-step reasoning for each task. The end result is a robust **multi-agent routing system** where Claude, GPT-4/Codex, Gemini, and other specialized tools work in concert, each focusing on what they do best, guided by a centrally defined playbook (the LangGraph).

Supabase Memory & RAG Layer

A cornerstone of this AI platform is its **memory and knowledge retrieval system**, backed by a Supabase (Postgres) database with vector search capabilities. This layer enables long-term memory of past tasks and conversations, and efficient retrieval of relevant information (documents, prior outputs, etc.) to augment AI prompts (Retrieval-Augmented Generation). We will design several database tables in Supabase to support this:

Database Schema: The main entities we need to store are **Tasks**, **Prompts**, **Sessions**, **Docs**, and **Embeddings**. Below is a summary of each table schema and its purpose:

Table Name	Key Columns & Types	Description
tasks	`id` UUID (primary key); `name` VARCHAR; `status` VARCHAR; `context` JSONB; `result` JSONB; `created_at` TIMESTAMP; `completed_at` TIMESTAMP; `user` VARCHAR; `tags` TEXT[]; `error` TEXT (optional)	Stores every task execution or queued task. Each entry corresponds to a task run or pending task, capturing input context and outputs. **Status** can be “pending”, “running”, “success”, “error”, “delayed”, etc. The **context** field holds the task inputs (parameters), and **result** holds the output or outcome (could be a result object or error info). We also record which user or agent initiated it and arbitrary **tags** (for categorization, e.g. `["chat","interactive"]` as seen in memory logs [27]). This table functions as both a **task log** and a live **queue**: tasks that need approval or retry remain in “pending” status. (In the current system, parts of this are split between an in-memory Celery queue, the `agent_inbox` table, and local logs; here we unify them). Whenever a task finishes, its row is updated with status and timestamps. This table enables querying task history, monitoring currently running tasks, and implementing retry/approval logic via status changes.

Table Name	Key Columns & Types	Description
prompts	`id` UUID; `name` VARCHAR; `template` TEXT; `model` VARCHAR; `variables` JSONB; `created_at` TIMESTAMP; `last_used_at` TIMESTAMP; `description` TEXT	Contains **prompt templates** or records of prompts. This table serves dual purpose: (a) store reusable prompt templates for various agents (each with a name/key, the text template, target model, and placeholders defined), and (b) optionally log actual prompt-completion pairs for analysis (in which case `template` would be the prompt content filled, and we might have another field for the completion). Primarily, it will be used for the former: to allow dynamic updates to prompts without code changes. For example, a row might have `name="CLAUDE_SOP_TEMPLATE"` and the template text with placeholders like `{company_name}`, and a short description like "Claude prompt for generating SOP documents." The **variables** field can list expected variables (keys) for this template. When tasks run, they can fetch the template by name from this table (or from the file system in `prompt_templates/`) – we'll likely sync the two or use one source of truth (for now, file-based templates are primary for version control, but syncing critical ones to DB allows on-the-fly tweaking via UI). If logging usage, we can update `last_used_at` each time a prompt is employed, which might help identify popular prompts or stale ones.
sessions	`id` UUID; `user` VARCHAR; `started_at` TIMESTAMP; `last_active_at` TIMESTAMP; `conversation_summary` TEXT	Tracks **chat sessions** or ongoing conversations. Each session can be associated with a user (or system if it's an agent-only session), and we record when it started and last active time. The session `id` will be attached to memory entries to link them to a particular conversation thread [27]. The optional `conversation_summary` can store a running summary or important facts of the session, periodically generated by an agent (to help compress context for very long conversations). This table helps implement multi-turn chats that persist across interactions – the frontend can start a new session or continue an existing one by passing the session ID to `/chat` endpoints, and the backend can then filter memory to that session. It also allows listing past chat sessions in the UI, resuming them, or cleaning up old ones.

Table Name	Key Columns & Types	Description
docs	`doc_id` UUID (primary key); `project_id` UUID; `title` TEXT; `content` TEXT; `author_id` VARCHAR; `created_at` TIMESTAMP; `updated_at` TIMESTAMP; `source_url` TEXT	Stores **knowledge documents or knowledge base entries** at a document level. This is for persistent knowledge that can be used in RAG. For example, if we upload a Markdown file or if Claude generates a product spec that we want to keep, it becomes a doc entry. The content might be the full text of the doc (for smaller docs), or this table might just store metadata (if docs are large, we might not store full content here but rather in chunks table). However, it's useful to have the full content for reference and maybe to re-embed if needed. `project_id` links to a project or category (there could be a separate **projects** table mapping project names to IDs, as seen in memory_utils where `projects` table was anticipated [28]). For example, project could be a brand or a topic area. `author_id` can indicate who created the doc (user, or which agent). `source_url` can store where the doc came from (if it was fetched from web or from an internal system). This table allows the UI to list all stored knowledge docs, display titles, authors, etc. It's essentially a document index.

Table Name	Key Columns & Types	Description
embeddings (or **doc_chunks**)	`id` BIGSERIAL (PK); `doc_id` UUID (FK to docs); `chunk_index` INT; `content_chunk` TEXT; `embedding` VECTOR(1536); `metadata` JSONB	This table stores the **embeddings for document chunks**. When a document is added or updated, we split it into chunks (e.g. ~200 words each) and generate a high-dimensional vector for each chunk using OpenAI's embedding model (text-embedding-ada-002) [29]. Each row here represents a chunk with its vector, and references which document it came from. We use Postgres's pgvector extension for the `VECTOR` type and have an index on it to enable similarity search. We can also store some metadata per chunk (for instance, `metadata` might include the section or heading it came from, or any tags assigned). The vector search RPC (`match_documents`) will likely operate on this table – e.g. find the top K chunks whose embeddings are closest to the query embedding, possibly filtering by project or doc if needed [30]. By separating this from the main docs table, we keep potentially large content and performance-heavy vector indexes separate from metadata. The system's RAG functionality queries this table when the AI needs context: given a user query or a task, we convert the query to a vector and find similar content chunks, then include those chunks (with maybe their doc title as context) in the prompt.

Additionally, the system already uses a **memory** table to log miscellaneous events and short-term memory; in this design, the `tasks` table largely covers task-related memory, and the chat transcripts would be covered by `sessions` + entries in `memory` or `prompts` (depending on how we log chat messages). We may still maintain a generic `memory` table for unstructured logs (as currently implemented [31] [32]), or we integrate it into the `tasks` / `prompts` tables by treating each user message or AI reply as a special task (type "chat_message") with input=message and output=response. However, to avoid confusion, we can keep a `memory` table just like now for quick logging of any event (with fields id, timestamp, type, content, etc.), and use **tags** to classify them (the existing system tags entries as "chat", "summary", etc. [27] [33]). This raw memory log can feed into summarizers or be pruned over time. In practice, we will have both: a structured tasks/prompts store and a raw memory log (we can implement `memory` as a *view* or simply continue using the existing approach but now with more structure).

Chunking & Embedding Workflow: When a new document is ingested (via an API endpoint or internally by a task), the system will: (1) create a doc entry in **docs** (assign a UUID, store title, etc.), (2) call the text splitter to break the content into chunks ~200 tokens (configurable) [34], (3) call the OpenAI Embedding API (using `OPENAI_API_KEY`) to get vector embeddings for each chunk [29], and (4) insert each chunk with its embedding into **embeddings** table. This process is already outlined in the current `memory_api.write_memory` and `doc_store.embed_and_store` functions [35] [36] – we will consolidate those into a single pipeline. We will ensure to attach metadata: for example, each chunk could

get a tag of the doc's title or an ID, and perhaps the chunk's section heading if we can parse it. The embedding vectors are stored in a column of type `vector`, dimension 1536 (for Ada embeddings). We'll set up a Postgres **extension (pgvector)** and a **similarity search function** (the Supabase RPC `match_documents`) that given a query text and optional filters will: compute the query's embedding (using the same API, possibly on the fly in a secure function or by the backend before calling the RPC), and then do `SELECT doc_id, content_chunk, similarity(embedding, query_vec) as score FROM embeddings WHERE project_id = X ORDER BY score DESC LIMIT K`. The RPC can encapsulate this, and our Python backend calls it for convenience [37]. If Supabase is unreachable or no results, we fall back to a basic keyword search in our local `doc_index.json` or in-memory list, as currently done [38] [39] (this is primarily for dev mode without Supabase).

Retrieval & Memory Usage: With the above in place, any agent that needs contextual knowledge can query for it. For instance, the Slack command `/brainops query <text>` can call an endpoint that uses `memory_store.search` or the vector search RPC to find relevant memory entries or docs containing `<text>` [40]. Those results are returned to the user or used in a prompt. For **conversational memory**, when a user sends a message to the AI (`/chat` endpoint), we retrieve the last N messages from that session from memory (or tasks table) to include as "Recent history" [23]. We can also retrieve any global context or profile (for example, if we have a stored "user preferences" doc, we might always fetch that as well). The value of storing memory in Supabase is persistence and queryability: the system can answer questions like "when was the last time we ran this task" by searching the tasks table, or "what was the summary of project X" by searching memory for tag "summary" and project X [41] [42].

Metadata Tagging: We make heavy use of **tags and metadata** in memory records to facilitate targeted retrieval. For instance, when the *Gemini memory agent* summarizes logs, it looks for memory entries tagged `"summary_candidate"` [43]. In our design, every memory or task entry can carry tags (like the type of content, project, etc.), and the search queries can filter by tags or time range [44] [45]. We will maintain conventions for tags: e.g. use `"chat"` for dialogue messages, `"task"` for task logs, `"error"` for error entries, `"summary"` for summarized content, etc. The **origin metadata** field (JSONB) stored with memory can include things like source (e.g. `voice` if it came from a voice transcription, as the current system does [46] [47]) or a link to an originating object (e.g. a `linked_transcript_id` or `clickup_task_id`). Our memory saving utility will merge such metadata when saving entries [48] [49]. This means, for example, if a voice message was transcribed and fed into the system, the memory of that event can carry `{"source": "voice", "tags":["transcript"]}`. Later, an agent could retrieve specifically transcripts by filtering on tags.

Maintaining Relevance: Over time, the memory table will grow. We will implement strategies like **time-based filtering** (only consider recent N entries by default, as currently done [25] with `memory_store.load_recent(scope)` which by default took last 5 entries, or memory_scope can indicate how far back to go), and **summarization** of older logs. The weekly strategy or summarizer tasks can compress old logs into a concise form and tag them appropriately (for example, a weekly summary log tagged `summary` that references the week's tasks). The RAG system can then pull from both detailed recent entries and higher-level summaries as needed.

Supabase Integration: To implement this, we ensure our Supabase client is properly initialized (with `SUPABASE_URL` and `SERVICE_ROLE_KEY` for full read/write) [50]. We'll use server-side RPC calls for vector search (to leverage Postgres speed), and direct table inserts/selects for normal queries (as currently

done with the Python supabase client [51] and in the code for logging tasks [7] and retrieving pending tasks, etc.). We will also implement database **constraints and triggers** as needed: e.g. a foreign key from `embeddings.doc_id` to `docs.doc_id` for consistency, and perhaps triggers to cascade delete embeddings if a doc is deleted or to update `updated_at`. We might also add a trigger to automatically vectorize certain short texts (for example, maybe vectorize each memory entry's `output` and store in an `embedding` column of `memory` table too) – but that could be heavy, so we likely vectorize only long-form knowledge docs and use text search for memory beyond that.

In conclusion, the memory & RAG layer ensures the AI agents are **"open-book" with our data**: they can recall what's happened (tasks, chats) and lookup what's known (documents, prior outputs) rather than working in isolation. This greatly improves coherence and accuracy. For instance, the system can answer "What was last week's priority review outcome?" by retrieving the memory entry for that task (since we tag it, or it contains "weekly priority review" in text) and either responding directly or giving it to an agent to formulate an answer. This design also means that adding a new knowledge source is straightforward: just insert docs and embeddings, and the agents automatically gain that knowledge through retrieval. Supabase provides a scalable cloud-hosted Postgres, and the use of pgvector means we have production-ready semantic search. By structuring the data well, we facilitate not only the AI functionality but also analytics (the team can query how many tasks ran, success rate, etc., using the tasks table) and compliance (we have logs of all AI outputs which can be reviewed if needed).

(Citations of code lines indicate where similar functionality exists in the current codebase for reference: e.g. saving memory [49], searching with tags and time filters [44], or requiring Supabase for persistent memory [1].)

Webhooks + Task Engine

The **Task Engine** is the component of the backend that receives events (from users or external systems), dispatches tasks to be executed (immediately or after approval), and manages task lifecycle (status, retries, notifications). It works in tandem with various **webhook endpoints** which act as entry points for external triggers. Here we outline how webhooks from Slack, ClickUp, Stripe, Make.com, etc., feed into the task system, and how the engine ensures tasks run reliably and update the outside world.

External Webhooks Integration:

- **Slack Webhooks:** We implement two types of Slack integration. First, a Slack **slash command** (e.g. `/brainops`) posts to our endpoint `/webhook/slack/command` [52]. This is used for operator control via Slack. For example, Slack text like `approve 12345` will call the endpoint; the backend verifies the Slack signature (`SLACK_SIGNING_SECRET`) and then interprets the command. In our system, we will parse the command to perform actions: `approve <id>` and `reject <id>` will change the status of a pending task in the tasks table and trigger its execution or cancellation, `status <id>` will look up a task's status, and `query <text>` will perform a memory search and respond with results [53]. The response can be sent back to Slack as a message (the endpoint will return a brief ACK and then use Slack Web API to post results asynchronously, possibly via a response_url or our own Slack bot token to reply). Second, Slack **event subscriptions** can be configured (e.g. message events). These would hit `/webhook/slack/event` – for instance, to automatically capture messages from certain channels as memory. The code already suggests storing Slack messages via `memory_store.save_memory` when events come in [54]. We will

maintain this: if configured, when Slack sends a message event (say someone tagged the bot or used a keyword), the system can log it or even trigger a task. Slack will thus be both an interface (for admins to command the system) and a data source (to feed info to memory or tasks).

- **ClickUp Webhooks:** ClickUp is used for project/task management. The integration will allow two-way sync: When our system creates or updates a task that corresponds to a ClickUp task, it uses ClickUp API (already partially implemented in `integrations.clickup_adapter`, and invoked after task success via `_maybe_sync_clickup` [55] [56]). Conversely, we can set up a webhook in ClickUp that calls our `/webhook/clickup` endpoint whenever a task in a certain list changes status (for example, an item moved to "Approved" list could trigger our automation). In the new routes, `integrations/clickup.py` will handle such payloads: verify a secret if available, parse the event (e.g. new task created or status change), then possibly enqueue a corresponding internal task. For instance, if a ClickUp task with name "Generate Proposal X" is created, we could automatically trigger our `autopublish_content` or similar pipeline with context from that task. Similarly, if one of our tasks requires manual steps (like human approval or additional info), we might create a placeholder task in ClickUp for tracking, which on completion triggers our webhook to mark as done. Essentially, ClickUp becomes an external UI for tasks—particularly for those who prefer a Kanban/To-do view—and our system ensures bi-directional updates (the code's `_maybe_sync_clickup` already handles creating/updating tasks on success). In practice, the operator could manage tasks from Slack or the Dashboard, and those can reflect in ClickUp for broader team visibility.

- **Make.com Webhook:** The platform supports integration with Integromat/Make.com by a generic endpoint `/webhook/make` [8]. This allows any automation scenario configured in Make to trigger our tasks. We will maintain this as a flexible ingress: the Make webhook likely expects a secret (like `MAKE_WEBHOOK_SECRET`) for authentication which we verify, then the payload might specify which task to run and with what context (similar to calling `/task/run`). Our `routes/webhooks.py` will parse the JSON and simply call `run_task` for the specified task, then respond with a task ID or status. For example, a Make scenario after processing a form submission could hit us with `{"task": "sync_sale", "context": {...}}` to run the sales sync automation. We'll ensure consistent response format and error handling (Make might require specific reply to acknowledge). The benefit is that our system can be integrated into larger workflows orchestrated by Make, but over time, as our platform grows, we might reduce reliance on external orchestrators and handle sequences internally with the agent graph.

- **Notion Webhook/API:** In the backlog [57], Notion integration is mentioned (import/export tasks). We will create endpoints like `/webhook/notion` or use the Notion API for two purposes: possibly ingesting tasks or content from a Notion database and outputting AI-generated results to Notion pages. For example, if content is drafted in Notion, our system could fetch it, improve it (via the SEO pipeline), and push it back. Implementation might involve periodic polling or a Notion integration configured to send events (Notion now has API but not sure about live webhooks, might need polling). In our design, we include a route `integrations/notion.py` to house functions like `import_tasks_from_notion()` or `update_notion_page(page_id, content)`. This could be triggered manually or by a schedule. It's not a primary feature initially but we have a slot for it.

- **Stripe Webhook:** Already present in current code, `/webhook/stripe` is used to handle events like a successful payment (new sale) [58]. On receiving a `checkout.session.completed` or similar event, we trigger the `sync_sale` task (notifying CRM, onboarding, etc.). We will keep this endpoint:

verify `STRIPE_WEBHOOK_SECRET` and then, for example, parse the event to get customer email or order info, then call `run_task("sync_sale", {...})`. The sync_sale task might itself call out to Make.com or other services as configured (the README mentioned a Make scenario for new sale [59], which might be replaced by direct API calls or an internal sequence). We ensure idempotency (Stripe may retry webhooks): our task engine could check if a sale ID was already processed to avoid duplication.

- **Other Webhooks/Endpoints:** The system also has endpoints for voice transcription upload (`/voice/upload`), status updates (`/webhook/status-update` presumably from external triggers), etc. We will maintain these in the new `routes/webhooks.py` or relevant route files. For instance, `/voice/upload` will accept an audio file, save it, maybe call Whisper (OpenAI's speech-to-text) to transcribe, store the text in memory, and possibly trigger downstream tasks (like automatically creating a task from a voice note). These features make the platform multi-modal. We'll also incorporate **email or scheduling triggers** if needed (e.g. a daily cron hitting an endpoint to run daily summary – though we prefer to do scheduling internally in the Task Engine).

Task Queue & Execution Engine:

Inside the backend, once a task request is received (whether via an API call from the frontend or a webhook), the Task Engine takes over:

- We use **Celery** with a Redis or RabbitMQ broker to offload long-running tasks from the main thread. The `/task/run` endpoint currently does `execute_registered_task.delay(...)` to enqueue a Celery job [60]. We will continue to leverage Celery for actual execution of tasks in the background, especially for those that might take many seconds (like calling Claude on a large prompt or waiting on an external API). For real-time interactive tasks (like streaming chat), we have the `stream_task` mechanism to stream via SSE without Celery (running directly async in the server process) [61] [62]. That will remain for tasks that support it.

- **Inbox & Approval Workflow:** Some tasks require human approval or review before execution (for example, tasks auto-generated by an AI plan but needing sign-off). The **agent inbox** is implemented via a Supabase table (currently `agent_inbox` or `task_queue`) that holds pending tasks with status "pending" [63]. In the new design, we may unify this with the `tasks` table (i.e. tasks table entries with status "pending" serve as the inbox). But we can still have a view or separate table if clarity is needed. The process is: when a task that needs approval is created (either by an agent like the weekly planner or manually by someone as a draft), it's stored with status pending and possibly a summary. The code already uses `agent_inbox.add_to_inbox(task_id, context, origin)` to insert an entry and even auto-summarize it using Claude (the `ai_inbox_summarizer` task) [64]. We'll keep that logic: e.g. if the weekly strategy agent generates a set of suggested tasks, they go to the inbox with a short summary of each so the human can quickly decide. The Slack command `approve <id>` or the Dashboard "Approve" button will call an endpoint (e.g. `POST /agent/inbox/approve`) which flips the status to "approved" and then triggers the actual execution (calling `run_task` on that task ID) [65] [66]. If rejected, we mark it rejected and perhaps log a note. This ensures AI doesn't execute potentially risky tasks without a human in the loop when desired. The inbox system will send notifications: if more than `INBOX_ALERT_THRESHOLD` tasks pile up, it can send a Slack push summarizing them (the current code `_maybe_send_alert` does this using

Claude to summarize multiple tasks into one message [67] [68]). We will maintain such features to keep operators informed.

- **Retry and Failure Handling:** The Task Engine has a built-in retry mechanism. In `run_task`, if a task's `run()` function throws an exception, we capture it and automatically queue the task into a `retry_queue` with incremented retry count [69] [70]. We have a separate task `task_rescheduler` that periodically checks for tasks whose retry is due or tasks that were delayed by user request. The `task_rescheduler.run` reads a local JSON of delayed tasks and for each that is due, it uses Claude to decide what to do: escalate, close, or re-run [71] [72]. In our design, we will formalize this by possibly using the database: tasks table has `status = 'delayed'` and a `resume_at` timestamp. A scheduler (could be Celery Beat or our own async loop) wakes up periodically, finds any delayed tasks whose time has come, and processes them. The processing can still consult an AI (Claude) as in the current design to decide next steps, which is a clever use of AI for ops (e.g. "This task was delayed 3 days ago, should we escalate it?" and Claude might answer "Yes escalate because it's important."). We will incorporate that logic, perhaps with more guardrails. The net result is that tasks rarely fall through the cracks: if failures happen, either the system retries automatically or prompts a human via Slack or changes status to "escalate" for manual intervention [73]. Each failure is logged (and Slack notifications for failures are sent immediately via `_log_task` setting level=error triggers a Slack message [74] [75]).

- **Event Bus / Messaging:** While Celery covers asynchronous job execution, for certain immediate propagations we might use an internal event system. For example, when a task completes, we might want to automatically trigger a follow-up task or send data to multiple places. We can implement this simply within the code (after a task finishes, in `run_task` we call some hooks like the ClickUp sync, Slack notify, maybe trigger an export). If complexity grows, we might formalize an internal pub-sub (even using Redis or a lightweight library) where different components can subscribe to events like "TASK_COMPLETED" with certain tag and react (e.g. a listener that if a task with tag 'publish' completed, it calls a deployment pipeline). Initially, though, our needs can be met with direct calls in code and scheduled checks.

- **Claude ↔ ClickUp Update Loop:** This refers to the interplay where Claude (or other agents) continuously update task status and content in ClickUp. For instance, consider a *weekly strategy* task: Claude generates a list of strategic action items and, via `_maybe_sync_clickup`, those get created as tasks in ClickUp [56] [76]. Later, as those tasks are completed or updated in our system, we reflect changes back to ClickUp. Conversely, if an operator updates something in ClickUp (like adding a note or closing a task), our webhook picks it up and could feed it to Claude for analysis (maybe to adjust its plan). This creates a loop: the AI plans tasks -> tasks show in ClickUp -> human maybe edits or adds details -> our system sees the update and perhaps triggers Claude to re-evaluate (for example, if a task was marked "blocked" in ClickUp, we could trigger Claude to analyze why and propose a solution). We will implement specific triggers for such loops. A simple approach: if a ClickUp webhook indicates a status change to a special status (say "Waiting for AI"), we call an agent to comment or progress it. Or if a task is marked "Done" by human, maybe we have Claude generate a brief retrospective (learning from success/failure) and store in memory. These kinds of continuous improvement loops ensure the AI is not just fire-and-forget but interacts smoothly with human project management.

- **Notifications:** Besides Slack, we could integrate other notification channels (email, MS Teams, etc.) as needed. The Slack integration is already robust (we have one central function `utils.slack.send_slack_message` used to push error logs and completions [77] [78]). We will keep using Slack for most internal alerts because it's convenient for the team, and possibly add an email summary (e.g. a daily email of what tasks ran – that could be a separate task scheduled daily, generating a report from tasks table and sending out).

- **Concurrency and Rate Limiting:** We will use Celery concurrency controls or FastAPI's dependencies to ensure we don't overload external APIs. The current code uses `slowapi` for rate limiting requests per minute [79] [80]. We will continue to use such middleware to protect endpoints (especially open ones like public contact forms) from abuse. For LLM API calls, we may implement logic to queue them if too many at once (the tasks architecture inherently queues and can be configured with concurrency limits per worker). We should also ensure one user doesn't spam a hundred tasks at the same time – rate limiting by user or globally via settings.

In essence, the Task Engine is the **heart** that connects **triggers → tasks → results → feedback** . It embraces a robust design: tasks are persistent objects in the DB (so we never lose track), all state changes are logged, and the engine uses both deterministic logic and AI assistance (like Claude summarizing or deciding escalation) to manage the pipeline. This combination of traditional programming (for guarantees and data integrity) with AI (for flexibility in decision-making) yields a resilient operations core.

Finally, every important event flows through this engine, allowing the **Dashboard UI and logs** to reflect real-time state. The `/dashboard/metrics` and related endpoints already aggregate counts of tasks, memory, errors [81] . In the new setup, those will draw from the unified tables (e.g. tasks succeeded/failed count from tasks table instead of multiple sources). This simplification will make building admin views (like "Inbox count" or "Tasks by status pie chart") straightforward.

🛠 Frontend Control Panel Blueprint

We envision a **two-part frontend**: an **Operator Control Panel** (BrainStackStudio Dashboard) for internal admins, and a **Copilot Interface** for end users of specific brands (e.g. MyRoofGenius Copilot). Both are built with Next.js and share a common design system and components where possible, but they serve different needs and audiences.

BrainStackStudio Operator UI (Admin Dashboard)

This is a secure web application for internal use (likely only accessible to authenticated team members) that provides full visibility and control over the automation system. Key features and sections of the operator UI include:

- **Task Inbox & Approval Panel:** A view showing pending tasks that require approval (the "Agent Inbox"). This might be presented as a list or Kanban column of tasks waiting for review. For each task, we display its summary (the short description generated by Claude when it was added to inbox [82]), origin (e.g. "recurring" or "user request"), time submitted, and options to Approve, Reject, or Delay. Approve triggers immediate execution (via an API call to `/agent/inbox/approve` with the task ID [65]), Reject marks it as rejected (with an optional reason that gets logged), and Delay lets the

admin specify a later time or condition (this will call an endpoint like `/agent/inbox/delay` with a payload of task ID and new time [83]). The UI for delay might have a date-time picker and possibly a dropdown of "fallback action" (our system uses a fallback field for delay which could be auto or notify [84]). The inbox panel will update in real-time (either via polling or WebSocket) as tasks get added or approved (we can leverage Supabase's real-time capability or simple periodic fetch).

- **Live Task Monitor & Logs:** A dashboard page that shows the current and recent activity of tasks. This can include a table of tasks with their status (running, succeeded, failed) updating live. We'll include columns like Task Name, Status, Start Time, Duration, and maybe a result summary or link to details. For tasks that are currently "running" or recently finished, the admin might click to expand details: see the input context, the output (or partial output if streaming), and any logs or errors. We will surface the logs that our backend collects (structured JSON logs) perhaps filtered by task ID. The UI might have a toggle for "Show last 24h" or filters by status and tag. This effectively surfaces the content of the `tasks` table in a friendly way. Additionally, a **Metrics panel** will display counts like total tasks executed, success rate, tasks by type (pie or bar chart for which tasks run most), and error rate over time [17]. These charts can use an internal metrics endpoint or compute from tasks table (Celery also emits metrics we capture like TASKS_EXECUTED, etc. which we can expose via `/metrics` for Prometheus [85]). We can use a library like Chart.js or recharts for this. The UI should also highlight if anything needs attention: e.g. a big red number of failed tasks today, or a notification if any task has been in "running" too long (possibly stuck).

- **AI Assistant (Copilot for Admin):** The admin dashboard will include an **AI chat panel** (similar to ChatGPT interface) that is *augmented with memory*. This is effectively **Copilot v2** endpoint that was mentioned [86]. In the UI, it looks like a chat: an input box where the operator can ask things like *"Summarize the recent deployments"* or *"Create a task to update the marketing site tomorrow"*. The backend `/chat` endpoint will handle these, using Claude or the default model, and return a streaming response with potentially **suggested tasks** in the answer [87] [88]. The UI will display the assistant's response token-by-token (we have a component `OutputStream.tsx` to append streaming text via SSE). If the response contains suggested tasks (the backend provides `suggested_tasks` list [89] [90]), the UI can highlight those or offer a button "Add to Inbox" for each. For example, if the assistant says "I suggest: 1) Run weekly priority review, 2) Prepare newsletter," the UI can let the admin click a suggestion to convert it into an actual task (calling perhaps `/task/generate` or directly enqueuing the named task). This way, the admin can have a conversational interface to manage the system – asking questions about memory (`/memory/search` can be triggered through such chat queries if the user says "search memory for X", as Slack query does) or instructing it at a high level, with the AI translating to tasks.

- **Memory Search & Knowledge Base:** A section of the dashboard will enable exploring the stored memory and docs. This could be a combined **search bar** where the operator enters a query and the UI shows results from both the memory logs and the knowledge docs. We will use the `/memory/search` or `/memory/query` API to fetch results [91] [92]. Results might be shown as snippets with context ("...found in Task 123 output on 2025-07-01" or "...in Document 'Onboarding SOP'"). The operator can click a result to see the full content (hence the **MarkdownViewer** component to display a memory entry or doc nicely, with highlights of query terms). For knowledge docs, we can allow viewing the entire doc, downloading it, or updating it. If an operator needs to add knowledge (say upload a new reference PDF or paste an SOP), we'll provide an **Upload Knowledge** interface (which calls `/knowledge/doc/upload` internally to embed and store it). This part essentially serves as an

internal wiki/search tool, backed by our RAG data. It's invaluable for verifying what the AI might use to answer questions and for curating the knowledge base.

- **Control Panel for Automation:** This is where various settings and triggers can be managed. For instance, the admin might have a UI to configure recurring tasks – a page showing the list of scheduled tasks (the `recurring_tasks.json` loaded by `recurring_task_engine`) and toggles or cron schedule editors for each [93] [94]. They could add a new recurring schedule via a form (select task, frequency daily/weekly, time, context params). The UI then calls an API (maybe `POST /tasks/recurring`) which adds to the JSON or DB and the engine picks it up. Another panel could show system status like environment info (if we expose `/diagnostics/state` [95], which might include current memory usage, Supabase connection status, etc.), and provide controls to flush caches or trigger certain maintenance tasks (like a "Run DB migration" button or "Re-index documents" which calls `/knowledge/index`). Essentially, this portion of the UI is for maintenance and fine-tuning – a replacement for manually running scripts or logging into servers. If needed, we can also surface environment variables (non-sensitive) to indicate which API keys are set or not (to quickly catch misconfigs).

- **Multi-Brand Management:** Since the backend serves multiple brands, the admin UI could allow switching context or viewing tasks by brand. For example, a filter to show only tasks for *MyRoofGenius* vs *TemplateForge* (assuming tasks or memory entries have a project/brand tag in context or origin). We might also allow certain actions by brand – e.g. deploying frontends or running brand-specific agents. Possibly a section listing each brand with some stats (how many tasks run for that brand today, is its front-end online, etc.). Because all brands share the same backend, this is mostly an organizational view.

- **Security & Feedback:** Provide UI for user management (for the admin login itself – though we may simply use Basic Auth or JWT with a single admin user as currently implemented [96] [97]). If multi-user, list of authorized users and roles (admin or viewer). Also a simple feedback form for admins to log issues/feedback into the system (calls `/feedback/report` to store a ticket [98]). That endpoint currently likely logs to Slack or Supabase. We can surface those reports on the dashboard as well (so the team can see known issues).

The admin UI is implemented as a Next.js app with likely a modern UI library (Tailwind + shadcn UI per README [99]). It will use JWT auth to call the backend (obtained via `/auth/token`, possibly stored as http-only cookie or localStorage plus a CSRF token for safety [100] [101]). Since it's mostly an internal tool, we value responsiveness and clarity: lots of real-time updates, possibly using **WebSockets or SSE** for events like new tasks or logs (Supabase Realtime could also push DB changes to the UI, but simpler might be a WebSocket from FastAPI that sends updates). We may implement a small Socket.IO or FastAPI websocket route streaming events that the UI can subscribe to (especially for log lines or new tasks, rather than polling). Alternatively, integrate with Supabase's realtime if feasible by listening on the tasks table changes.

MyRoofGenius Copilot UI (User-Facing)

MyRoofGenius (and similarly TemplateForge or other brand-specific interfaces) is a more focused application meant for end users or customers. It will expose only certain capabilities of the system, in a domain-specific and user-friendly manner. Features for MyRoofGenius Copilot might include:

- **Conversational Q&A Assistant:** A chat interface where the user (e.g. a roofing salesperson or a homeowner client) can ask questions related to roofing estimates, solar installations, etc. This is backed by our system's knowledge base and tools. For example, a user might ask "Can you generate an estimate for a 2000 sq ft roof replacement in Denver?" The Copilot UI sends this as a request (perhaps to a brand-specific endpoint like `/copilot/query` that internally might format some context like default model and knowledge base project). The system might then run a pipeline: fetch relevant pricing data from knowledge, run `generate_roof_estimate` task if needed, or directly answer if general. The UI will present the answer, possibly with a PDF or detailed breakdown if produced. This chat would be more constrained than the admin one – likely it won't expose task suggestions or anything technical. If complex multi-step needed, the backend handles it behind the scenes. The UI just sees question -> answer.

- **Form-Driven Task Submission:** For tasks that are clearly defined (like "Generate EagleView report parsing" or "Create proposal document"), the UI can present a form rather than a free text chat. For instance, an **"Estimate My Roof" form** could ask the user to upload an EagleView JSON or enter some parameters, then on submit call the `parse_eagleview_report` task followed by `generate_roof_estimate` task automatically. The results (CSV of quantities, cost breakdown) would be shown or downloadable. Another form might be "Request Solar Installation Guide" which triggers the generation of product docs. These forms make it easier for non-technical users to use the AI without phrasing a request. Under the hood, the submission calls our API (with proper brand context and user auth) and then the UI polls the `/task/status/{id}` until ready (or uses WebSocket events to know completion).

- **RAG-Enabled Search for Users:** If we want to empower users with self-service information retrieval, we can include a **Knowledge Base Search** component. E.g., TemplateForge might let users search a library of templates or AI-generated articles. The front-end could call the same `/memory/query` or a brand-filtered variant and display relevant documents or answers. Potentially, an LLM could be used to answer user queries by pulling from docs (like a ChatGPT on documentation). We might incorporate a simplified version: user asks a question in Copilot chat, the backend uses RAG with brand-specific docs to answer. The UI can show the answer along with "Source: TemplateForge Guidebook" or similar, building trust through citations.

- **Live Updates and Notifications:** If some tasks take time (like generating a detailed proposal might take 1-2 minutes if it's doing heavy analysis), the user UI should handle that gracefully. Possibly by showing a loading state with messages like "Crunching numbers for your estimate…", and then either streaming partial results or notifying when done. Since we might not want to expose the concept of "task IDs" and manual refresh to users, we can use WebSockets or server-sent events to push the result. For example, upon form submission the API could respond immediately "task accepted" and the UI opens an EventSource on `/task/status/stream/{id}` that sends progress or completion. Or simpler, the API keeps the connection open and streams the final output (though that's not typical if it takes a long time; better to decouple with a push).

- **User Authentication and Session:** Depending on context, MyRoofGenius Copilot might require user login (if it's for internal use by salespeople) or could be publicly accessible (with limited functionality). If login is needed, we'll integrate with Auth – possibly using the same JWT system (we can have user role tokens separate from admin tokens). The backend settings allow multiple users with roles [102], so we'll use that. The UI will have a login page if required and obtain a token from `/auth/token`. If public, perhaps no login but certain sensitive actions wouldn't be available.

- **Branding and Customization:** The Copilot UI will carry the branding of MyRoofGenius (logo, colors, etc.), even though it's powered by BrainStackStudio backend. TemplateForge's UI would look different accordingly. We maintain them as separate Next.js apps or as one app with dynamic theming (but separate simplifies deployment and customization). They will however use common components for chat, form, etc., possibly imported from a shared library to reduce code duplication.

- **Limits and Guidance:** For a good UX, especially if the audience is non-technical, the UI will guide the user on what they can do. For example, showing a list of example questions or tasks (like "Ask: What is the best material for a 2000 sq ft roof?" or a button: "Generate my roof report"). These can be static hints or dynamic based on memory (e.g. if user has some data stored, suggest related questions). Also, any limitations or disclaimers should be shown (like "Estimates are based on standard pricing and may not reflect actual quotes").

- **Real-time Collaboration:** Possibly out-of-scope initially, but one can imagine multiple users or an admin monitoring the copilot's conversations. If needed, the system's memory and session model allow an admin to see what a user asked and what the AI responded (since all interactions are stored). This might be surfaced in admin UI rather than user UI, but it's a benefit of the unified backend.

Overall, the user-facing Copilot focuses on **simplicity and task-specific UX**, whereas the admin dashboard focuses on **breadth and control**. Both, however, rely on the same backend APIs. The Next.js apps will use `NEXT_PUBLIC_API_BASE` to know the base URL of the API (empty if same domain or a specific URL if separate) [103] and include the auth token for requests (for admin, a cookie or header; for user, maybe similar or none if open). They will also share certain pages like the marketing site (the BrainStackStudio admin app might double as the public site for marketing, as mentioned in README with routes `/products`, `/services` etc. for public content [104]). We can maintain that by having the admin app serve public pages and protect the `/dashboard` path for actual app, or similarly.

Frontend–Backend Interaction: All UI operations correspond to backend endpoints:

- Approving tasks -> calls `/agent/inbox/approve` (with CSRF token included in header because our backend protects non-GET with CSRF for browser clients [105] [106]).
- Submitting a new AI task -> calls `/task/run` or specialized endpoints (`/task/nl-design`, `/chat`, etc.) depending on UI context.
- Chat streaming -> opens a connection to `/chat` with `stream=true` to get SSE events [107].
- Searching memory -> calls `/memory/search` or `/memory/query` as appropriate.
- Uploading file -> calls `/knowledge/doc/upload` (for knowledge) or other specific endpoints (like `/voice/upload` if we add voice on user side).
- Logging in -> calls `/auth/token` with credentials to get JWT cookie [97].

- The PWA aspect: the admin dashboard includes a manifest to allow "Add to Home Screen" on mobile [108]. We will ensure to include that and service worker for offline caching of last data, which is nice for mobile monitoring.

By using SSR or static generation carefully, we can ensure the public pages are SEO friendly (for marketing content), while the app pages use client-side auth. Given Next 13's App Router, we might do server-side rendering for some pages. It's not critical for the internal app except maybe for initial load performance.

UI Technologies: We stick with **Tailwind CSS** and **shadcn/ui** (a popular library of accessible components built on Radix UI) to get a consistent and modern look. We will use **Framer Motion** for subtle animations (as seen in home page snippet [109]). The UI should be responsive (so operators can even check things on mobile via the PWA, which is mentioned in README as well [108]).

Summary of Components and Pages:

- **Components**: Reusable building blocks.
- `CopilotHeader` (maybe brand logo and menu – we saw `frontend/components/CopilotHeader.tsx` in repo likely for the user side).
- `PromptPanel` – a text box with maybe a dropdown to choose model (Claude vs GPT vs Gemini) for admin, or hidden default for user.
- `OutputStream` – displays the streaming text with a typing indicator.
- `MarkdownViewer` – uses a library to render markdown to HTML, maybe with styling (plus could add copy buttons for code blocks, etc.).
- `TaskList` – lists tasks with icons or status colors; admin version might have interactive controls, user version might just show what they've run.
- `Charts` – if using a chart library, embed charts for metrics.
- `SearchBar` – with results dropdown or redirect to a search results page.
- `UploadButton` – for knowledge upload, maybe integrated with drag-drop.

- **Pages (Admin)**:

- `/dashboard` – main dashboard overview (key metrics, recent tasks, maybe a welcome message).
- `/dashboard/inbox` – tasks pending approval.
- `/dashboard/tasks` – full list of tasks (with filters by date/status).
- `/dashboard/tasks/[id]` – detail view of a specific task (show input/output and log).
- `/dashboard/agents` – perhaps a visualization of agent graph or controls (optional).
- `/dashboard/memory` – search memory UI.
- `/dashboard/docs` – list knowledge docs, with view/upload options.
- `/dashboard/settings` – system settings like schedules, etc.
- `/dashboard/chat` (or integrated via a chat widget on all pages) – the admin copilot chat.

- (Public pages like `/` home, `/products`, etc., which can be static content as per marketing need).

- **Pages (MyRoofGenius)**:

- `/` – might directly be the Copilot interface (or a marketing page explaining it).

- `/copilot` – chat interface if not on home.
- `/estimate` – a form for roof estimates.
- `/reports` – maybe list of past reports generated (if we allow users to retrieve old results, might require login or a session).
- `/guide` – knowledge base Q&A page if needed.
- `/about` etc – brand info.

We will ensure **security**: the admin pages will check for admin role JWT (and the backend already has dependency `require_admin` on certain routes [102]). The user pages will either not require auth or check for user token. Also implement CSRF protection for forms (the backend sets a CSRF cookie and expects header, which our frontend will handle by reading the cookie and including header for state-changing requests, as configured [110]).

To tie it together, these frontends will empower both internal team and external users to leverage the AI platform effectively. By providing a friendly UI on top of a complex backend, we hide the complexity and present clear value: For admins, a command center for AI operations; for users, a helpful assistant or tool that solves their domain-specific problems (whether generating content or insights).

Prompt Template Architecture

Effective prompt design is vital to guiding the AI agents' behavior. We will establish a **prompt template architecture** organizing all prompts into structured files and categories, with variables and formatting rules that ensure consistent, high-quality outputs. The prompt templates will be stored in the repository under `tasks/prompt_templates/` (as noted in the file structure), and possibly mirrored in the database (the **prompts** table) for runtime flexibility.

Archetypes of Prompts: We identify several archetypal prompt categories, each with a distinct purpose and style:

- **Claude Markdown Templates:** These prompts are designed for Anthropic's Claude to produce **well-structured Markdown documents**. Examples include:
- *SOP (Standard Operating Procedure) Template:* A prompt that instructs Claude to draft a procedural document with step-by-step instructions, given context about a process. It might include placeholders for the process name, roles, tools used, etc., and emphasize a clear, numbered list structure.
- *Blueprint Template:* Used when we want Claude to generate a technical design or project plan (like this very blueprint). It would have sections like *Executive Summary, Objectives, Architecture, Tasks, Implementation Plan*, etc. The prompt might say "You are an expert system architect. Please create a detailed design document in Markdown with the following sections: ..." and we fill in bullet points to cover or context to incorporate. Variables could include system name, requirements, and any provided context (like backlog items or constraints).
- *Changelog Template:* Instructs Claude to draft release notes or a changelog from a list of changes. It might have a structure like version number, date, and list of changes categorized into added/changed/fixed. The prompt ensures the output is in proper Markdown (using lists, maybe bold headings for each version). We can supply the raw commit list or change summary as input to this template.

- *Bundle Template:* If we want Claude to output multiple related documents in one go (e.g. a "bundle" of content pieces), this template guides how they should be combined. For instance, a product launch bundle might include a blog post, a tweet thread, and an email draft all in one output. The template might explicitly delineate sections for each piece, with clear markers (like "Blog Post: \n<blog content>\n\nTweet Thread:\n1. ..."). This ensures the output is easy to split if needed. Variables can be the product details, angle, etc.

All Claude markdown templates will share some common style guidelines: we will instruct to use appropriate Markdown syntax (for headings, lists, tables), to not exceed a certain length unless allowed (Claude can handle large output, but we might set expectations), and to include certain metadata if needed. **Metadata tags** can be embedded as HTML comments or YAML front matter. For example, we might include a YAML front matter at the top of an SOP like:

```
---
title: "Onboarding SOP"
author: "Claude AI"
date: "2025-07-14"
tags: ["SOP","HR"]
---
```

This could be used later for conversion to PDF or indexing. Or we might put an HTML comment like `<!-- AUTOGENERATED: DO NOT EDIT -->` to mark it as AI-generated. These conventions will be documented and consistent so that any downstream process (like publishing pipelines) can detect them.

- **Codex Handoff Template:** This prompt is for instructing the Codex (GPT-4) agent how to generate code from a spec. It serves as the bridge between Claude's output and Codex's input. We want to eliminate guesswork, so this template will be very explicit. For instance, a template file `codex/implement_code.md` might read:

```
You are an AI coding assistant. You will be given a software specification in
Markdown, enclosed in <spec> tags. Follow the specification exactly to produce
code.

<instructions>
- Only produce valid code and nothing else (no explanation).
- If the spec contains multiple files, output them in the required format (as
JSON or as separate code blocks with file names).
- Preserve all structure as described.

</instructions>

<spec>
{{{ blueprint_markdown }}}
</spec>
```

```
Begin coding now.
```

Here `{{{ blueprint_markdown }}}` (using triple braces to indicate raw insertion without escaping in Jinja, for example) will be replaced by the actual Claude blueprint content. The instructions emphasize that the assistant should not deviate. We might decide a specific output convention: perhaps Codex should output a JSON with keys as filenames and values as code (which our system then writes to files). Or as Markdown with fences labeled by filename as earlier. We have to pick one and train Codex via the prompt to stick to it. JSON might be easier for automated parsing (we can include in instructions "output JSON with keys 'filename' and 'code'"). This template thus ensures Codex knows the expected format and context. Additional variables could be model or language specifics (if we know programming language, but likely the spec itself includes that).

- **Gemini SEO Template:** A prompt tailored for Google's Gemini model, focusing on SEO or content optimization. For example, `gemini/seo_optimize.md` might contain a prompt like:

```
You are an expert content editor and SEO specialist. Improve the following
content for search engine optimization and clarity, without changing its
meaning.

Focus on:
- Using the keyword "{{ target_keyword }}" naturally throughout.
- Adding an engaging meta description (50-160 characters).
- Ensuring headings are meaningful and include relevant phrases.
- Making the tone appropriate for {{ audience }}.

Content:
"""
{{ original_content }}
"""

Now provide the improved version of the content, in Markdown format. Also
include a suggestion for a meta description at the end, under a heading "Meta
Description".
```

Variables here include `target_keyword`, `audience`, and the `original_content`. The template ensures the output includes what we need (the optimized content and a meta description). We instruct usage of Markdown (so if the original had formatting, it remains). By having a dedicated template, if we want to tweak how we instruct Gemini (maybe we find it needs more guidance on not altering certain things, or including an H1 title), we can adjust in one place.

We might have other Gemini-oriented templates as well, e.g. a **Summary template** for the `gemini_memory_agent` which summarizes logs. Currently, the code constructs the prompt dynamically

[42], but we could externalize it: e.g. `gemini/brief_summary.md` saying "Summarize these logs briefly: {{ logs_text }}". This keeps consistency.

- **Perplexity Research Injection Template:** When we combine search results into Claude's prompt, we want a template that formats it cleanly so Claude can incorporate it. For instance, `research/injected_answer.md` might look like:

```
You are a knowledgeable assistant with access to research data. Using the
information below, answer the query comprehensively.

Query: "{{ user_question }}"

Research findings:
{{#each sources}}
- "{{this.text}}" (Source: {{this.source}})
{{/each}}

Provide a well-structured answer. Cite sources by number when used.
```

(Above uses Handlebars-like syntax for iteration – actual implementation could flatten into a single string.) Essentially, we list the findings (maybe with numbering or some indicator so Claude can refer to them). We instruct Claude to use them in the answer and cite. The output might then have references like "[1]" corresponding to the given source list. This template ensures that our merging of Perplexity results and question is systematic, not ad-hoc.

Another template in this category might be for *automated report writing from data*, if we had search or stats. But main idea is to make sure the assistant reads the provided info.

- **Other Prompt Types:** We also have smaller prompts like for the *inbox summarizer* (Claude prompt that summarizes pending tasks to include in Slack alert [68]) or the *delay rescheduler* prompt (Claude decides escalate or not [71]). These can be templated too (to adjust phrasing or criteria easily). We can put them under a category like `claude/ops/` or similar if we like. For clarity, one prompt per file with a descriptive name.

Structure & Variables: Each template file will be written in plain text (likely Markdown or text with placeholder syntax). We will adopt a templating language, likely Jinja2 (since Python can easily fill Jinja templates using our `utils.template_loader.render_template` as seen used [111]). For Jinja2, variables are denoted `{{ var }}` and we can have simple control structures. The template files might have `.md.j2` extension to indicate Jinja, or we strip extension. In our `template_loader.py`, we can load the file and do `Template.render(fields)` to produce the final prompt. This allows separation of logic and prompt wording.

We will maintain consistent variable names and pass in a dictionary from tasks. For example: - Claude SOP task will call something like `render_template("claude/sop.md", {"procedure_name": ..., "steps": ...})`. - The Codex handoff pipeline will call `render_template("codex/implement_code.md", {"blueprint_markdown": spec_text})`. - In code, these templates are

referenced by name (and maybe we maintain a mapping if needed). The **prompts table** in DB can also store them: e.g. a row with name "codex_implement_code" and content matching the file. We could synchronize on startup (load all files into DB for quick editing in UI if needed). However, initial approach is file-first for version control.

Formatting Rules for Outputs: We instruct in prompts how output should be formatted, but we'll also document guidelines: - All AI-generated docs should be valid Markdown for easy preview and conversion. Use `#` for main titles, `##` for sections, etc. Use backticks for code, etc. - If the output is code that will be saved to file, prefer to output just the code (or use the agreed JSON/markdown format and we'll parse it). - We discourage including any extraneous prose in Codex output (like “Here is the code:” – the prompt explicitly says not to). - For multi-part outputs (like the bundle), separate clearly (maybe with level-2 headings or markdown comments). - Ensure each prompt asks the AI to stay within scope and not include info it wasn’t given. E.g. in summarizer prompts, instruct not to hallucinate beyond provided logs. - Use delimiters in prompts (like triple quotes, XML-like tags, or fenced blocks) to clearly show the AI what the content is vs instructions. (We see this used in many open prompts to avoid prompt injection issues). - Make sure to include examples in the prompt if needed. For very critical formats, an example can help. For instance, in the Codex template, we might actually include a short dummy spec and the expected JSON output as an illustration. But that lengthens prompt; might not be needed if we phrase well.

Metadata for Conversion: As mentioned, for certain outputs like PDF conversion, including metadata in the output helps. E.g., if we output a documentation bundle that will be turned into a PDF, including title, author, date at top is useful. We might have the template itself insert those from variables (like project name, current date, etc.). Alternatively, we add them after generation (the `docs` table can combine metadata). But doing it in template ensures it's part of what AI sees and produces nicely.

For example, the SOP template might automatically do:

```
# {{ title }}

*Last updated: {{ date }}*

{{ content_body }}
```

So all SOPs have a standardized title and timestamp. Or a blueprint might list "Prepared by Claude on <date>".

We will also incorporate special markers for post-processing. If we plan to generate PDFs via a tool, maybe we want page breaks or image placeholders. We can instruct AI to include something like `<div style="page-break-after: always;"></div>` if needed to separate sections for printing, though that’s an edge case. At minimum, consistent heading levels and perhaps a table of contents (Claude can generate if asked).

To ensure these rules are followed, we rely on prompt clarity. If we find the outputs deviating, we adjust the template and maybe add explicit "Do not do X" instructions.

Organization & Naming: Each file in `prompt_templates` folder uses lowercase and underscores or hyphens. The categories can be subfolders or part of name. We proposed subfolders by agent or use-case. Example file paths: - `prompt_templates/claude/sop.md` - `prompt_templates/claude/blueprint.md` - `prompt_templates/claude/changelog.md` - `prompt_templates/claude/bundle.md` - `prompt_templates/codex/implement_code.md` - `prompt_templates/gemini/seo_optimize.md` - `prompt_templates/gemini/brief_summary.md` - `prompt_templates/research/injected_answer.md` - `prompt_templates/ops/inbox_summary.md` (for summarizing inbox tasks) - `prompt_templates/ops/task_delay_decision.md` (for Claude deciding on delayed task action)

We'll document the purpose of each template at the top of the file in a comment, and ensure tasks refer to them. This way, non-developers or the team's prompt designers can edit these templates without touching Python logic – making prompt iteration faster.

Finally, we will keep track of **versions** of prompts if needed. Possibly just via git history or adding a version number in a comment. If a prompt heavily influences outputs and we update it, we might want to note it in the changelog.

The Prompt Template Architecture thus gives a **library of "AI instructions"** that is as important as our code. Just as code modules can be reused, prompt templates are reused by tasks. This separation also allows future fine-tuning: e.g., if one day we fine-tune a model or we have a different model requiring slightly different phrasing, we could have alternative templates, selected by model name or version.

In summary, by categorizing prompts into Claude's document-style prompts, Codex's coding prompt, Gemini's SEO and summarization prompts, and injection templates for research, we ensure each agent is *prompted in the optimal way*. The structure enforces consistency, so outputs of the same type always have the same format (less surprise when integrating results). And embedding metadata in outputs (like titles, dates, citations) ensures these AI-generated documents are immediately useful in context like publishing or reporting, with minimal post-editing.

💻 Codex Task Blueprint

This section enumerates all the new or modified code components (Python modules and TypeScript/TSX files) that need to be created to realize the design, serving as a "to-do list" for implementation. Each item includes the file path (within the monorepo structure), a brief description of its purpose, and key functions or classes to implement. This comprehensive list will guide the Codex agent (or developers) in generating the necessary code with no guesswork.

Backend (FastAPI & Tasks) Files:

File Path	Purpose	Key Functions/Classes
`apps/backend/main.py`	FastAPI application setup and entry point. Importantly, mounts routers, configures middleware (rate limiting, CORS, logging), and triggers startup events (like DB init).	`app = FastAPI(...)` with lifespan; `include_router(...)` for each module; `startup_event` to ensure DB migrations a template loading.
`apps/backend/routes/tasks.py`	Defines API endpoints for running tasks and pipeline flows (non-chat). Handles immediate or queued execution and streaming responses.	Endpoints: `POST /task/run` (enqueue Cele returns task_id) [60] ; `POST /task/design` nl-design` (invoke design pipeline Claude-> `GET /task/status/{id}` (check Celery or status, returns result) [10] ; potentially `POST /pipeline/run` (generic pipeline tri name).
`apps/backend/routes/auth.py`	Authentication endpoints (token issuance, refresh, logout). Manages JWT and cookies.	`POST /auth/token` (verify user from `AUTH` and return JWT + CSRF) [97] ; `POST /auth/re` (rotate JWT) [113] ; `POST /auth/logout` (clea [114] .
`apps/backend/routes/memory.py`	Endpoints for memory and knowledge base operations (RAG).	`POST /memory/query` (semantic search ac [12] ; `POST /memory/write` (store new doc embed) [115] ; `POST /memory/update` (re-en updated doc) [116] ; `POST /knowledge/doc/` separate, to handle file upload of docs; calls w `GET /memory/search` for simple text searc memory (calls memory_store.search with que filters).
`apps/backend/routes/webhooks.py`	Consolidated external webhook endpoints (Slack, ClickUp, Stripe, Make). Verifies secrets and delegates to integration handlers or tasks.	`POST /webhook/slack/command` (parse co text and call appropriate function) [52] ; `POST /webhook/slack/event` (log Slack e memory or trigger action) [54] ; `POST /webho clickup` (handle ClickUp event: e.g., new tas automation); `POST /webhook/stripe` (ver signature, on payment success trigger `sync_ task) [117] ; `POST /webhook/make` (verify `MAKE_WEBHOOK_SECRET`, run incoming task `POST /webhook/notion` (if using, handle N events). These handlers likely call into `integ *.py` utility functions or directly `run_task`

File Path	Purpose	Key Functions/Classes
`apps/backend/routes/agents.py`	Endpoints for agent-specific actions and multi-agent orchestrations.	`POST /agent/inbox/approve` (approve a task) [65]; `POST /agent/inbox/reject` (re maybe not explicitly in current code but we a `POST /agent/inbox/delay` (delay task wi time) [83]; `POST /agent/inbox/prioritiz` or summarize inbox) [118]; `GET /agent/inb` pending tasks, possibly with summary counts `POST /agent/inbox/summary` (force Clau summarization of inbox, if needed). Also endp initiate specific multi-step flows like `POST /a plan/weekly` (calls weekly strategy agent) similar endpoints already present which will r LangGraph internally.
`apps/backend/agents/langgraph.yaml`	YAML configuration defining the agent graph nodes and flows (as described in the LangGraph section).	*No functions* (data file). Will include node defin type, model, etc.) and flows with edges. This f parsed at runtime to build the graph.
`apps/backend/agents/base.py`	Core logic for executing the LangGraph. Includes a parser for the YAML and an executor that can run a given flow by traversing nodes. Manages context passing and conditional logic.	Class `AgentNode` (with properties like id, typ reference or model name, etc.); Class `Agent` `FlowExecutor` with method `run_flow(flow_name, inputs)`. The exe lookup node definitions from YAML (or a pre- structure) and call either LLM wrappers or int functions accordingly. Also, functions like `load_graph_config()` to parse YAML on s Handles condition evaluation and possibly pa execution if specified.
`apps/backend/agents/claude_agent.py`	Wrapper for calling Claude API (Anthropic). Provides standardized interface (e.g., taking a prompt and returning the completion). Also might handle streaming.	`def run(prompt: str, model: str = "` `v1") -> str` to do a single completion call `httpx` to Anthropics endpoint) [121] [122]; `as` `stream(prompt: str) -> AsyncGenerato` for streaming SSE tokens (if Anthropic stream available via API, otherwise simulate chunkin keys from settings. Possibly refactor from exis `claude_prompt.py` logic.

File Path	Purpose	Key Functions/Classes
`apps/backend/agents/codex_agent.py`	Wrapper for OpenAI GPT-4 (or 3.5 Codex if still separate) focusing on code generation.	`def run(prompt: str, model: str = "` `> str` – calls OpenAI ChatCompletion API w prompt and system instructions (like few-sho Ensure it respects tokens limit and format (pc instruct model to output JSON if needed). `as` `stream(prompt: str) -> AsyncGenerat` needed for streaming code (though likely we and return full code because partial code isn' without all context).
`apps/backend/agents/gemini_agent.py`	Wrapper for Google Gemini model calls. Uses the PaLM API (GenerativeLanguage API) as in current `gemini_prompt.py` [123] [124].	`def run(prompt: str, model: str =` `"gemini-1.5-pro") -> str` – posts to Go current code does (with API key) [125], handles and returns content. Possibly handle chunkin too long (1000-token limit maybe). We might specialized calls like a `def embed_text(text: str) -> List[f` Gemini or PaLM offers embedding (if not, we OpenAI for that, which belongs elsewhere).
`apps/backend/agents/search_agent.py`	Logic for performing web searches or calling Perplexity API. If no official API, this might call our own headless or use an alternative (maybe we use an SERP API or a local knowledge base of recent data). However, given the mention of Perplexity, likely a function that triggers an external process or uses a stored search mechanism.	`def run(query: str) -> dict` – for nov just log the query and return {"status": "queu `perplexityRelay.ts` stub [126]. Ideally, int Perplexity's API if available or use a service lik Google Custom Search to get top results, the LLM summarize. Perhaps break into two: `search_web(query) -> list[Source]`, `summarize_sources(sources) -> str`. Claude or GPT to summarize multiple results. node might thus orchestrate internally (calls s then calls an LLM agent to summarize) unless that to the graph definition (which could inste two nodes: search and Claude). For now, a sir implementation: use some search client (if no placeholder).
`apps/backend/tasks/__init__.py`	Task registry initializer. Scans all task modules and registers them (similar to current `brainops_operator._load_tasks`) [4].	On import, execute function to import submo call `register_task`. Possibly maintain `_TASK_REGISTRY` dict mapping task_id to f may not need a dataclass like `TaskDefinit` table stores metadata, but keep for quick loo

File Path	Purpose	Key Functions/Classes
`apps/backend/tasks/autopublish_content.py`	Example of a *composed task* that might use multi-agents under the hood to publish a blog or product content. (From README: publish an article to site, trigger Make uploads, send newsletter) [127]. We might reimplement it to use our new pipelines: e.g. uses Claude to draft content, maybe Codex to format HTML, etc., then calls integration to Make or direct API.	`TASK_ID = "autopublish_content"`; `do` `run(context)` – parse context (article detai call `agents.base.run_flow("publish_co` `context)` if we define that flow, or manually orchestrate steps: e.g. call Claude for draft (`claude_agent.run(template="blog_po` `fields={...})`), call an integration to publ (maybe using `requests` to a WordPress or then return result. This task demonstrates us multiple components.
`apps/backend/tasks/generate_product_docs.py`	Task to create product documentation with Claude and push to documentation site. Possibly uses a prompt template (Claude) and then uses an integration (like calling GitHub or an API to commit the doc).	`run(context)` – context might have produ doc format needed. It will use something like `claude_agent.run(template="product_` `fields={...})`, get markdown output, the `integrations.notion.publish_page(ou` if docs site is in git, create a file via GitHub AP we might have an integration). Return succes doc.
`apps/backend/tasks/parse_eagleview_report.py`	Task that parses a JSON (EagleView roof report) into a CSV. This might not involve AI at all (pure Python logic). We'll implement it as standard code.	`run(context)` – read JSON from context (n context has file path or JSON string), parse fie compute quantities, output a CSV file content structured data. No external calls. Ensure this fairly quickly within Celery. Return perhaps a data (maybe store file in Supabase storage or
`apps/backend/tasks/generate_roof_estimate.py`	Task that calculates material and labor costs from roof quantities (again more deterministic, but could involve some AI if we want narrative). Likely straightforward calculation given some parameters or using known rates.	`run(context)` – context might include are type, pitch, etc. The task fetches standard pri config or memory (maybe stored in docs or a environment), multiplies out costs, and return breakdown (could return as structured dict or formatted Markdown/CSV). Possibly it could c to format the estimate nicely in a paragraph, heavy lifting is arithmetic.
`apps/backend/tasks/claude_prompt.py`	(Refactored) A generic task to prompt Claude with given input. We might not need this as a task separate from agent wrapper, but current code has it to provide a way to call Claude via task queue or CLI [128]. We'll keep a simple one for debugging.	`TASK_ID = "claude_prompt"`; `run(con` expects `context["prompt"]` and optional `context["template"]`, calls `claude_age` with that and returns completion [122]. Useful Claude responses or making it accessible via `run`.

File Path	Purpose	Key Functions/Classes
`apps/backend/tasks/nl_task_designer.py`	A task that takes a natural language goal and generates a sequence of tasks (essentially, the initial step for auto-generating an automation pipeline). The current system had something similar (maybe this is how `/task/nl-design` is supposed to work). We'll implement it to utilize Claude or GPT-4 to output structured tasks.	`TASK_ID = "nl_task_designer"`; `run(` – context has a `goal` description. The task u prompt like "Given the goal, list a set of tasks to achieve it." Possibly uses the `chat_to_pr` similar chain present (the current code used `chat_to_prompt` to suggest tasks from a u message [129]). We might directly call Claude w special template to generate JSON: e.g. tasks of {task: ..., depends_on: ...}. Then return that further processing (like automatically creatin tasks in inbox).
`apps/backend/tasks/ai_inbox_summarizer.py`	Task to summarize pending tasks in the inbox (used for notifications when many tasks waiting) [130]. We will keep this to offload summarization.	`TASK_ID = "ai_inbox_summarizer"`; `run(context)` – context might include the fetch pending tasks within. Calls Claude or Ge produce a summary sentence like "Tasks: X (d W)...". Already in current code uses `claude_prompt.run` with "Summarize inb We'll adjust to use a proper prompt template summarization.
`apps/backend/tasks/claude_output_grader.py`	Possibly a task to review or grade Claude's output (maybe in the works to have GPT critique itself). If not needed immediately, skip or implement if referenced in issues. If needed: could have Claude or GPT-4 evaluate quality of outputs (for QA pipeline).	If implementing: `TASK_ID = "claude_output_grader"`; `run(context` has some output to grade and criteria. It retu or feedback. This might be used internally to run needed.
`apps/backend/tasks/recurring_task_engine.py`	Manages recurring tasks scheduling (reads a list of schedules and enqueues tasks when due) [93] [94]. We'll expand if needed or keep as is but integrated. Possibly convert storage to DB (but can keep JSON for simplicity).	`run(context=None)` – checks current time `recurring_tasks.json`, for each due, cal `agent_inbox.add_to_inbox` to queue it ('recurring') [131]. Functions: `add_recurring_task(entry)` to add to JS save [132] (wired to an API). We might add `remove_recurring_task` or an update fu completeness.

File Path	Purpose	Key Functions/Classes
`apps/backend/tasks/task_rescheduler.py`	Handles delayed tasks and escalation [133] [71]. We will use this to process tasks that were postponed. It uses Claude to decide action. We'll keep the logic, maybe adjusting to mark statuses in the `tasks` table instead of JSON where possible.	`run(context=None)` – loads `delayed_ta` checks each if due [134], for due ones calls Cla `claude_prompt.run`) with prompt asking close/rerun [71]. Then depending on decision, task (calls `run_task` synchronously or enqu mark as closed or escalate (update status). Al agent_inbox.mark_as_resolved or update stat accordingly [72]. We might adapt escalate to j pending but mark differently (maybe "escalat triggers human attention in UI).
`apps/backend/tasks/memory_diff_checker.py`	(Possibly referenced by endpoint `/memory/audit/diff` [135]). Could be a task to find differences between memory states, maybe for debugging. If in scope, implement a simple diff on memory logs; otherwise, deprioritize.	If implemented: would fetch two sets of mem (maybe from two time ranges or sessions) an diff (which could be done by a git-diff library comparing text). Not critical for core function possibly skip unless needed.

Integrations & Utilities:

File Path	Purpose	Key Functions/Classes
`apps/backend/integrations/slack.py`	Utilities for sending messages to Slack and verifying Slack requests. Already present as `utils.slack` in current code [77] [78]. We maintain here.	`def send_message(text: str, channel=None)` – posts to Slack incoming webhook or bot (using `SLACK_WEBHOOK_URL` for simplicity as in current code [78], which sends errors to Slack). `verify_request(request)` – use `SLACK_SIGNING_SECRET` to validate signature in headers (Slack sends timestamp and signature, we combine and compare hash). Also parse Slack command payload format.

File Path	Purpose	Key Functions/Classes
`apps/backend/integrations/clickup.py`	Functions to call ClickUp API (create task, update task) and perhaps handle incoming webhook payload. The code already hints with `create_clickup_task` and `update_clickup_task` in clickup_adapter [136]. We'll implement these properly with ClickUp API endpoints.	`def create_clickup_task(list_id, title, description, token)` – uses ClickUp API (POST to `/api/v2/list/{list_id}/task`) with given token. Returns created task ID. `def update_clickup_task(task_id, fields, token)` – PUT to update name/description etc. `def handle_webhook(payload)` – parse incoming JSON from ClickUp webhook and return a context or call a task (like if a task moved to "To Automate" list, trigger something).
`apps/backend/integrations/notion.py`	Functions for Notion integration. Possibly create/read Notion pages or databases. If we have an API key (`NOTION_API_KEY`, `NOTION_DB_ID` in settings [137]), we can use the official Notion SDK or direct HTTP. Use-case: maybe pushing generated docs to Notion.	`def create_page(title, content_markdown, parent_db)` – create a new Notion page (with given parent database or page ID) and fill with content (Notion API requires structured blocks; might need a Markdown->blocks conversion or simple if only text). `def update_page(page_id, content_markdown)` similarly. If receiving webhooks from Notion, `def handle_webhook(payload)` – likely not, since Notion doesn't send outbound webhooks easily; we may rely on polling if needed (or just user manual trigger).
`apps/backend/integrations/make.py`	A simple handler for Make.com. Likely just verifying a secret and packaging data to feed to tasks. Make usually can send whatever; here we design expecting maybe `task_name` and `context`.	`def handle_webhook(request_data)` – e.g., if `request_data` has `task` and `context`, simply do `run_task(task, context)`. Could allow multiple tasks or some branching if needed. The endpoint logic might suffice without separate function, but we place here for clarity.
`apps/backend/integrations/stripe.py`	Verify Stripe webhook signature and extract event data to call tasks. Possibly use Stripe's library if available, or manual HMAC check on payload using `STRIPE_WEBHOOK_SECRET`.	`def handle_webhook(request)` – get headers and body, use `stripe.Signature` if using library, or manually construct signature to compare. Determine event type; if `checkout.session.completed`, call `run_task("sync_sale", {"session_id": ..., "customer_email": ...})`. If `invoice.paid` or others, maybe handle similarly or ignore.

File Path	Purpose	Key Functions/Classes
`apps/backend/core/settings.py`	Already defined environment settings using Pydantic [138], we may extend to include any new config (like model names, or feature flags for multi-agent mode).	Add fields if needed: e.g. `OPENAI_MODEL` default, `ENABLE_CHAIN_MODE`. But largely we use existing keys: `CLAUDE_API_KEY`, `OPENAI_API_KEY`, `GEMINI_API_KEY`, etc. Possibly add e.g. `PERPLEXITY_COOKIE` or similar if needed to call it. Ensure `model_config` has correct env file path.
`apps/backend/core/scheduler.py`	A new module to manage background jobs (cron-like tasks). If we don't use Celery Beat, we can implement a lightweight scheduler using `asyncio` loops on startup.	Could define an `async def scheduler_loop()` that runs in background (FastAPI lifespan event) checking every minute for tasks to run: call `recurring_task_engine.run()`, call `task_rescheduler.run()`, etc. Or use Celery periodic tasks (in `celery_app.py` currently might be configured, but simpler is to have our own loop or rely on an external cron hitting certain endpoints). We'll likely implement at least a basic internal scheduler for recurring tasks (since we store them in JSON, easier to manage directly).
`apps/backend/core/security.py`	Utilities for hashing passwords (using `passlib` pbkdf2 as in main [139]), verifying JWT tokens, etc. Possibly refactor logic from main into here.	`def hash_password(password)`, `def verify_password(password, hash)`. JWT: `def create_access_token(sub, roles, expires_minutes)` and `def decode_token(token)`. Right now, main does JWT encode/decode inline [140] – we can move that here for cleanliness.
`apps/backend/core/logging.py`	Set up loggers, sinks (Slack, Supabase). Already partly in main where it adds Slack and supabase sinks to loguru logger [78] [141]. We modularize that.	`def configure_logging()` – remove default handlers, add JSON stdout, add `_slack_sink` for level ERROR (posting to Slack) [78] [141], add `_supabase_sink` if supabase available (inserts log into `logs` table) [142] [143]. This allows tracking errors in Supabase as well for later analysis. Possibly, integrate with an APM if decided (not in scope here).

Frontend (React/Next.js) Files:

File Path	Purpose	Key Components/Functions
`apps/frontend/brainops-admin/pages/dashboard/index.tsx`	Admin Dashboard home – an overview page with charts and recent activity.	Use `Charts` component to display metrics (e.g., tasks executed today vs week). Use `TaskList` to maybe show last 5 tasks or pending tasks summary. Possibly a welcome message or quick actions (buttons like "New Task", "Run Daily Plan"). Fetch data from e.g. `/dashboard/metrics` for counts [144] or directly from tasks endpoints.
`apps/frontend/brainops-admin/pages/dashboard/inbox.tsx`	Inbox page showing pending tasks needing approval.	Fetch pending tasks from `/agent/inbox` (which can return list of tasks with summary) [63] . Render list with each task name/summary, and "Approve / Reject / Delay" buttons. On approve, call `/agent/inbox/approve` with task_id [65] ; on reject, maybe call a similar endpoint (or we treat reject as mark resolved with status 'rejected'); on delay, open a modal to pick time and call `/agent/inbox/delay` [83] . Use WebSocket or polling to update list when changes occur (or optimistic UI updates removing approved task).
`apps/frontend/brainops-admin/pages/dashboard/tasks.tsx`	Page listing all tasks (or perhaps tabs for active, completed, failed).	Call `/tasks?status=...` (we might implement a query or just fetch all from `/dashboard/full` if exists [145]). Display in a table with pagination if large. Columns: ID, name, status, user, created_at, completed_at. Perhaps color-code status. Each row clickable to see details.
`apps/frontend/brainops-admin/pages/dashboard/tasks/[id].tsx`	Task detail page, showing context and output of a specific task, plus logs.	Fetch `/tasks/status/{id}` for current status and result [10] , and maybe another endpoint for full detail (if context is large, might store in DB and we retrieve). Show context (could JSON.stringify nicely or formatted if known fields). If output is text or markdown, render via `MarkdownViewer` . If it's an error, show error stack trace. Also display logs: we could call `/logs/task/{id}` if we had, or reuse `tasks.result` if it contains error info. Include a "Re-run task" button maybe (which could enqueue a new one with same context).

File Path	Purpose	Key Components/Functions
`apps/frontend/brainops-admin/pages/dashboard/memory.tsx`	Interface to search and browse memory logs.	Provide a search bar. On submit, call `/memory/search` with query and optional filters (date range, tags, user). Display results as a list of snippets: e.g., if result is memory entries, each might have `output` or `input` text. Use `MarkdownViewer` for any that are markdown. If result includes metadata like source or tags, display those. Possibly allow filtering by tag via checkboxes (e.g., chat vs task vs error). If nothing entered, maybe list recent memory (last N entries from `/memory/search?limit=50`).
`apps/frontend/brainops-admin/pages/dashboard/docs.tsx`	List all knowledge documents in the system's knowledge base (the docs table). Allow upload new doc and search within docs.	Fetch list from `/knowledge/doc/list` (we may implement or use `/memory/query` with no query but project filter). Display titles and maybe first line. Provide upload form (file or copy-paste): on submit, call `/knowledge/doc/upload` with file content and metadata; refresh list when done. Each doc item clickable to view details. Also a search input that specifically queries docs: calls `/memory/query` with `project_id` if needed (or an endpoint that returns doc search results in context).
`apps/frontend/brainops-admin/pages/dashboard/docs/[id].tsx`	Document detail page, showing full content of a knowledge doc. Also possibly allow editing and re-saving (update).	Fetch `/knowledge/doc?id=...` (we may add an endpoint to get full content by id, or the list could have content truncated). Render content via `MarkdownViewer`. If editing allowed for admins: either a simple textarea with markdown or maybe a rich text (for now, maybe markdown editor). On save, call `/memory/update` with doc_id and new content [116], then re-render (the backend will re-embed it asynchronously or in that call). Show list of related docs or an option to delete doc (call an endpoint to remove doc and its embeddings).

File Path	Purpose	Key Components/Functions
`apps/frontend/brainops-admin/pages/dashboard/agents.tsx`	(Optional) If we want to visualize or let admin tweak the agent graph. Could show the LangGraph structure (nodes and flows). Might not be core, but could be read-only view for trust.	If implemented: fetch `langgraph.yaml` (maybe the backend can serve it as JSON via an endpoint). Display nodes (maybe a simple list or network graph using a library). Possibly highlight active flows. Not a priority though – could skip in initial pass.
`apps/frontend/brainops-admin/pages/dashboard/settings.tsx`	Admin settings panel for configuring system parameters (like thresholds, schedules, user management).	This depends on what we allow to change at runtime. Could show the `recurring_tasks` list: fetch from an endpoint or directly from JSON via an endpoint that returns it. Provide a form to add a new recurring task (task_id, frequency, time). Also list existing with enable/disable or remove. Could show current environment flags (like which models keys are configured). For user management, list users (from env or an in-memory store since we have `AUTH_USERS` as env – might not be editable from UI easily unless we allow adding to environment via secrets endpoint). This could also hold a form to send test notifications or flush logs, etc. We must be cautious exposing too much; maybe keep minimal: recurring tasks and a link to supabase (since DB changes beyond that likely via code).
`apps/frontend/brainops-admin/components/TaskList.tsx`	Reusable component to display a list of tasks (with minimal info). Could take props like tasks array and a subset of fields to show. Used on dashboard home and tasks page.	Renders a table or list of tasks. If used on home, perhaps only incomplete tasks or recently completed. Accept onClick handler for row (to view detail). Possibly use an icon or label for status (like a green check, red x, etc.).

File Path	Purpose	Key Components/Functions
`apps/frontend/brainops-admin/components/PromptPanel.tsx`	A panel for inputting a prompt to the AI (Claude/GPT/Gemini) along with controls. Admin version may allow selecting which model or chain to use.	Contains a textarea for the prompt text. Might have a dropdown or toggle for model (e.g., “Claude vs GPT-4 vs Gemini” vs an "Auto" setting). Also possibly a dropdown for mode (Chat vs Design vs Execute Task). But perhaps simpler: admin can pick from available flows: e.g. "Brainstorm", "Ask Data", "Dev (Claude+Codex)". This selection could adjust how we call backend (maybe hitting different endpoints or including a parameter). On submit, it calls the appropriate endpoint (`/chat` if just Q&A, or `/task/nl-design` if design mode, etc.) with `stream=true`. It then creates an `OutputStream` component below to display result. It should also handle any suggested tasks from the response (if `ChatResponse.suggested_tasks` comes back [89]). If suggestions exist, display them as clickable chips " Deploy update" etc., and on click, maybe open a modal to confirm adding to Inbox or run immediately.

File Path	Purpose	Key Components/Functions
`apps/frontend/brainops-admin/components/OutputStream.tsx`	Component to display streaming output from the backend. It connects to an EventSource (SSE) and appends tokens as they arrive. Also indicates when done or if error.	Use React state to accumulate `content`. On mount, open `EventSource` to a given URL (passed via props or if we use fetch that yields a readable stream, but SSE is simpler in browser). For `/chat` streaming, backend yields `data: token` lines [146]. We parse and append `token` to content. If the event stream closes, we finalize. Also possibly handle `suggested_tasks` which might not come until the stream is done (in current logic, suggestions are only available after full completion [147]). But since backend does memory_store.save_memory in finally with suggestions, maybe suggestions are not pushed via SSE. Alternatively, we could adapt to send a final SSE event with suggestions encoded (like a JSON blob). Simpler: after stream ends, do one fetch to `/chat/to-task` with the user message to get suggestions (the system already has an endpoint for chat to tasks conversion [148]). But the code already calls `chat_to_prompt` internally and included suggestions in ChatResponse for the non-stream case [88]. For streaming case, in our design maybe we close SSE and then call an endpoint to get suggestions. We'll handle as needed. The component should scroll as content grows (like auto scroll to bottom). Possibly allow copying the output. If output contains markdown, we could either display raw text (and let MarkdownViewer interpret it after done) or even live-interpret partial markdown – probably simpler to display raw during streaming then replace with formatted once done. For chat, raw text is fine. For other outputs that are code or multi-part, might need special handling if needed (e.g., if an output is code to be saved, we might not stream it but deliver at once).
`apps/frontend/brainops-admin/components/MarkdownViewer.tsx`	Renders Markdown content as HTML in a safe way.	Likely use a library like react-markdown or Marked to parse and render Markdown content. Enable plugins for code syntax highlighting, etc. We should also handle if the markdown includes HTML or dangerous stuff; use allowed list of tags or trust it if from our AI (should be fine mostly). Ensure it has styling (maybe via Tailwind prose classes). This component is reused anywhere we need to show markdown (knowledge docs, task outputs that are markdown, etc.).

File Path	Purpose	Key Components/Functions
`apps/frontend/brainops-admin/components/Charts.tsx`	Displays charts for metrics. Could use e.g. Chart.js via react-chartjs. Render possibly multiple small charts: tasks per day line chart, success vs fail donut, etc.	Functions to transform metrics data from API to chart datasets. If `/dashboard/metrics` returns JSON counts [81] (like tasks_executed, tasks_failed, memory_entries, etc.), we can display key numbers as KPIs. If we want timeseries, might require the backend to give a series (we might instead query tasks with grouping by day, perhaps out of scope initially). Possibly just showing cumulative counts and maybe tasks by status pie from the tasks list. We will start simple due to time. The component encapsulates the library details.
`apps/frontend/brainops-admin/utils/apiClient.ts`	Utility for calling backend API with proper headers (auth and CSRF) and handling SSE if needed.	`function apiGet(url)` wraps fetch GET with `Authorization: Bearer token` if token in storage and include `credentials: 'include'` for cookies if needed (since we set JWT as cookie perhaps). Similarly `apiPost(url, data)`. Also a helper `openEventSource(url, onMessage)` for SSE connections. Possibly handle refresh token logic if 401.
`apps/frontend/brainops-admin/utils/auth.ts`	Manage authentication state in the front-end. e.g., retrieving token from cookie or localStorage, and login flow.	`function login(username, password)` calls `/auth/token` and if success, stores token (maybe in memory or cookie is already set HttpOnly by backend which is safer). If we rely on HttpOnly cookie, then `apiClient` should send those cookies automatically. We might not even need to expose token to JS. But we do need CSRF token for modifying requests (provided by `/auth/token` response as `csrf_token` in JSON [149] and also set as cookie by backend). We can store that CSRF token in a JS variable for later use (e.g., set as header `X-CSRF-Token` via fetch). So auth util can after login store CSRF token in e.g. sessionStorage. Also include `logout()` that calls `/auth/logout` and clears state. Possibly `useAuth` hook for React to protect routes, but since Next can do server-side redirect if no token cookie, simpler to handle on client mount.

File Path	Purpose	Key Components/Functions
`apps/frontend/myroofgenius/pages/index.tsx` (or /copilot.tsx)	Main user interface for MyRoofGenius Copilot. Likely a combined chat + form interface or at least an introduction and a chat start button.	Could show a welcome message ("Hi, I'm the Roof Genius Assistant! Ask me anything about your roof or request a report.") and then an input for a question. Possibly directly embed the PromptPanel and OutputStream but simplified. Also maybe quick action cards like "Get a Roof Estimate" that navigates to /estimate page.
`apps/frontend/myroofgenius/pages/estimate.tsx`	A form specifically to generate a roof estimate. It might allow user to upload a report JSON or enter dimensions manually.	Contains input fields (area, roof type, etc.) or a file upload for EagleView JSON. On submit, calls the backend: possibly first `parse_eagleview_report` if file provided, then `generate_roof_estimate` (or we have one endpoint that orchestrates both). We might create an endpoint `/task/webhook` or similar that accepts a generic trigger with two tasks sequentially [150]. Alternatively, we call `/task/run` twice and show intermediate result. But better to encapsulate logic server-side. For user simplicity, one click yields final output (maybe a download link or a nicely formatted cost breakdown on screen). The page shows a loading indicator while processing (subscribe to status via SSE or poll). Once done, display results: we could present a table of materials and costs, plus a summary sentence. Maybe also allow downloading CSV or PDF (perhaps the tasks returns a URL for a CSV in storage or includes CSV as text).
`apps/frontend/myroofgenius/pages/guide.tsx` (optional)	Possibly a knowledge base Q&A page if end-users can query stored docs (like a FAQ powered by the AI).	Provide a search or question box. When submitted, call a backend route that does RAG Q&A on user's behalf (maybe the same `/chat` but restrict model to use knowledge base). The result is shown, along with citations if any (could show sources from the answer if formatted with [1][2]). This is like a self-help portal. If not needed, skip implementing.
`apps/frontend/myroofgenius/pages/about.tsx` etc.	Static pages for marketing (About, Contact).	Mostly static content or a simple contact form (which could call our backend `/api/contact` that sends to Make webhook as per README [151]). Likely these come with the template or we create simple content.

File Path	Purpose	Key Components/Functions
`apps/frontend/myroofgenius/components/ChatBox.tsx`	A simplified chat interface for user. Combines a text input and message display. Unlike admin, user gets a one-on-one conversation style feed (with user and assistant messages).	It will manage a conversation state (list of Q&A). On user submit, it appends user message to list, then calls backend (likely `/chat` with their message and perhaps a fixed model or a session_id tied to user if we want context retention). It handles streaming the assistant response (similar to OutputStream, but we may integrate it here for simplicity). Once done, it appends the assistant answer to chat. This chat might also automatically incorporate RAG (the backend's `/chat` already includes memory and knowledge search). The UI shows each message in a bubble format. Also might allow resetting the conversation (new session id).
`apps/frontend/myroofgenius/components/EstimateForm.tsx`	The form component used on estimate page for input.	Handles the fields and file upload, and on submit triggers the appropriate backend calls. Possibly separate from page logic for reusability. It might either directly call an API that handles everything, or piecewise: if file present, first upload to `/memory/relay` or a special endpoint to store file content (maybe we should add an endpoint to accept file and run parse in one go). We may have to handle file reading in JS (FileReader to JSON string) and include in context. Simpler: instruct user to obtain JSON (maybe not realistic; better to accept file). We'll likely add in backend something like `/webhook/eagleview` just for this use-case, or extend `/task/run` to accept a file by first uploading to memory and referencing it. For now, implementing in UI: if file provided, do one request to upload file to our backend (we have `/voice/upload` example, we could similarly have `/file/upload` for general files which stores in Supabase storage or memory log), then get an ID or content and include that in the `context` for tasks. Could also do FileReader in browser to get text and send in JSON (embedding raw file content in JSON is possible if not huge). We'll decide based on expected file size. Possibly simpler: instruct user to paste JSON text into a textarea. We can accept moderately large JSON text (a couple thousand lines maybe). For first iteration, that might be acceptable.

File Path	Purpose	Key Components/Functions
`apps/frontend/myroofgenius/components/ResultDisplay.tsx`	A component to show results of estimate or similar in user-friendly way (graphical or structured).	For example, if `generate_roof_estimate` returns a structured breakdown (like a dict of items with costs), we can format a nice table: "Material X: $Y, Labor: $Z", etc., and a total. Or if it returns markdown text, just render via MarkdownViewer. Provide a "Download CSV" button if CSV data available (e.g., if parse task provided CSV). That button could create a blob from CSV string and use `URL.createObjectURL` to download, or if backend gives a file URL, just link it.
`apps/frontend/myroofgenius/utils/apiClient.ts`	Similar to admin's apiClient but possibly a bit different if authentication is different. If the user side is open access (no login), we might not need JWT except maybe for rate-limiting. If any user auth (maybe for internal sales team), we might reuse the admin auth system with separate user credentials. We'll plan as if open for now.	If open: just handle base URL and common headers (like `Content-Type: application/json`). If auth needed (the user might log in to see say past reports), then similar to admin but perhaps restricted scope. Possibly separate environment config for user app (NEXT_PUBLIC_API_BASE used here too).
`apps/frontend/myroofgenius/utils/format.ts` (optional)	Helpers to format data for display (e.g., currency formatting for cost, etc.).	`formatCurrency(amount)` etc., to ensure consistency in displays like estimate results.

Deployment & CI Files:

File Path	Purpose	Key Configurations
`.github/workflows/ci-cd.yml`	GitHub Actions pipeline automating testing and deployment of both backend and frontend.	Steps: on push to main, checkout code, set up Node (for frontend) and Python (for backend) environment. Run `turbo run lint && turbo run test` to lint and test all apps. Possibly run `pytest` for backend (assuming tests exist). Build Docker image for backend (or use heroku/Render deploy) and deploy to Render (could authenticate via Render API or use a GitHub integration). For frontend, either build and deploy to Vercel (if using Vercel, linking repo) or build static and publish to S3/Netlify. Alternatively, separate workflows for backend and frontend. Also perhaps a job for syncing prompt templates to DB (if needed) or running migrations. Include secrets in GitHub for API keys (though likely these stay in Render/Vercel). This file ensures any Claude/Codex generation artifacts (like if we use an AI step in CI, which is unlikely for now) are integrated.
`render.yaml` (in repo root or under infra/)	Render.com configuration if needed (though they might allow using Dockerfile directly or web settings). The README mentions a provided render.yaml [152]. We'll include it to specify the environment variables and startup commands for the backend service.	Define service: likely `name: brainops-backend`, `plan: free` or starter, `env: python`, `buildCommand: pip install -r requirements.txt && alembic upgrade head`, `startCommand: uvicorn apps.backend.main:app --port 10000` (matching how to run). Mount secrets for API keys. For static dashboard, either have backend serve it (we already do by copying build to static/) or deploy the static as a web service (but since admin can be served by backend at `/dashboard/ui`, we do that). We may also define a CRON job on Render if needed to call our endpoints for recurring tasks (if we didn't implement internal scheduler fully – but we plan to).

File Path	Purpose	Key Configurations
`vercel.json` (if deploying user frontend to Vercel)	Configuration for Vercel deployment of Next.js app(s). It might specify rewrites if needed or environment. Possibly not needed if Vercel auto-detects. If user app uses some serverless function for contact form, might configure that.	Might contain routes for any backendless functions, but likely our user app will call the main backend for everything, so no need special. If hosting multiple apps, might have to define each. Alternatively, we host admin also on Vercel (but we prefer static via backend). The user app likely can be static or minimal SSR. We'll skip heavy config unless necessary.
`Dockerfile` (for backend)	Dockerfile to containerize FastAPI app with all dependencies and runtime. Provided in repo root (we saw one in repo). Likely mostly fine but we ensure it's updated with new structure.	Use a Python base (slim), copy code, install requirements, set `ENV UVICORN...`, etc. If we also containerize frontend, might separate or let Vercel handle frontend. Probably simpler: one Docker for backend including static admin UI. The user UI could be static on Vercel or built and served under a different path. Perhaps just let Vercel handle user UI for scale and domain (myroofgenius.com).
`docker-compose.yml` (for local dev)	Compose file to run backend, possibly a Postgres (for Supabase emulator or direct PG), and maybe a Redis for Celery in dev. Also could run the Next.js dev servers if needed.	Services: db (Postgres with pgvector extension loaded), maybe adminer or pgweb for DB UI, redis for Celery broker, backend (build from Dockerfile or use volume mount for live reload if using uvicorn reload), and optionally a service for supabase or vector DB if needed. Not strictly necessary if using Supabase cloud in dev, but good for offline. This helps new devs set up quickly.
`tests/` (various backend tests)	Python tests to verify critical functionality (we likely extend `tests/test_basic.py` to cover multi-agent flows etc.).	For example: test that `memory_store.save_memory` inserts into supabase properly (could mock supabase), test `recurring_task_engine` schedules tasks correctly, test prompt templates produce expected structures (maybe by unit testing `template_loader` or small prompts). Also integration tests for endpoints (using FastAPI TestClient, e.g. test /task/run returns task_id and /task/status eventually gives result). Given complexity, focus on at least core logic tests.

File Path	Purpose	Key Configurations
`sdk/index.ts` & `sdk/services/DefaultService.ts`	TypeScript SDK for interacting with our API – perhaps used by the Next.js apps or possibly offered for third parties. It's listed in repo. We'll update if needed to reflect new endpoints.	Probably have classes or functions mapping to each endpoint (like `getTasks()`, `runTask(task, context)` etc.). If admin and user apps are within monorepo, they could import directly rather than use network – but since API is separate service, better to call via HTTP. SDK might not be heavily needed internally, but if it exists, update endpoints accordingly.

Each of these files will be implemented following the blueprint specified. The objective is to have a clear mapping from design to code artifacts: for example, when adding the multi-agent orchestrator, we know to create `base.py` and `langgraph.yaml`; when exposing new endpoints, we know exactly which route file to edit or create. This structured task list ensures that Claude (for writing documentation and templates) and Codex (for generating the code files) have an exact blueprint to follow.

By breaking the implementation into these discrete files and responsibilities, multiple agents (or developers) can work in parallel. For instance, Codex can generate the FastAPI router code while another instance generates the React components, all based on this synchronized plan. We have listed even small utility files to leave nothing ambiguous. The end result will be a cohesive system where every component aligns with a documented purpose, and the chances of missing functionality are minimized.

Finally, after code generation, we will perform thorough testing (both automated and manual) to validate that each piece interacts correctly: e.g. creating a task via UI goes through API to Celery to execution and returns result to UI; multi-step pipelines produce the intended outcomes; memory search returns relevant info, etc. This blueprint and task list serve as the contract for that implementation phase.

CI/CD & Deployment System

To ensure smooth and reliable releases of the BrainStackStudio platform, we will set up a **CI/CD pipeline** and deployment architecture that covers testing, security, and automated deployment to our chosen hosting services (e.g. Render for backend, Vercel for frontends). The CI/CD process will tie together Claude's documentation outputs and Codex's code outputs by verifying that everything is consistent and passing tests before deploying.

Continuous Integration (CI): Every change to the repository (especially to the prompt templates or code) will trigger a GitHub Actions workflow: - The workflow will run on pull requests and merges to main. - **Install & Lint:** It will install Python dependencies and Node packages, then run linters/formatters (e.g. `flake8` or `black` for Python, `eslint` for JS/TS) to enforce code quality. Any style issues cause the build to fail early. - **Run Tests:** Next, it executes backend tests (`pytest`) and frontend tests (`npm test` for each app if we have any). For backend, we may use a test Supabase URL or a local Postgres (we can spin up a service container with Postgres and provide it to tests). Our tests should cover core logic as outlined. Code coverage can be measured to track improvement over time. - **Validate Templates:** We could include a step to verify that prompt templates are valid (maybe just a simple check that no forbidden patterns or placeholders missing – possibly using a script). For example, ensure no template has undefined

`{{ variable }}` by rendering them with dummy data. - **Build Artifacts:** If tests pass, the pipeline will build the production artifacts. For the backend, that means building the Docker image. For frontends, generating static builds. Using Turborepo, we can have a step like `turbo run build --filter=...` for each app. - **Security Checks:** We will also enable dependency vulnerability scanning (GitHub does some by default, and we can use tools like `pip-audit` or `npm audit`). Also possibly run a container scan on the built Docker image. - **AI Integration Checks:** Although not typical, since our development involves LLMs, we could incorporate a step where, for example, a prompt blueprint is fed to a validation script or even a dry-run with a small model. However, this might be out of scope for automated CI given cost and variability. We instead rely on unit tests for logic and human review for prompt content.

The CI ensures that **Claude-ready docs and Codex-generated code remain in sync**: For instance, if a developer manually changes code without updating docs or vice versa, tests might fail or at least the difference would be noticed in code review. We can also set up a rule that the architecture spec (this document or a derivative in `docs/architecture.md`) must be updated if certain core files change (this can be enforced lightly via PR template reminding to update docs if architecture changes).

Continuous Deployment (CD): Once CI passes on the main branch: - **Backend Deployment:** We use Render.com for hosting the FastAPI service (as indicated by README). The pipeline can automatically deploy by pushing the new Docker image or triggering Render via API. Render can auto-deploy on new commits if configured, reading `render.yaml`. We'll ensure `render.yaml` is updated to include any new env vars and that migrations run (maybe via a start command or a separate job). The environment variables (API keys, etc.) are stored in Render's dashboard, not in code, to keep them secure. We set `ENVIRONMENT=production` in Render so that settings enforce required keys [153]. After deployment, Render will start the server and the dashboard should be reachable. We include a **health check** endpoint (like `/metrics` or a dedicated `/health`) that Render can ping to ensure service up. - **Frontend Deployment:** For the BrainOps admin dashboard, since it's static (Next.js exported to static files) and is served by the backend at `/dashboard/ui`, we actually need to update those static files on the backend. One approach: the Docker build process could run `npm run build && npm run export` for the dashboard_ui and include the output in the image. If we keep them separate, we can host admin UI on Vercel or Netlify as static. But embedding in backend is convenient for internal access control (the backend already expects to serve it with auth). We'll likely bundle it with backend deployment. That means our Dockerfile will have a step to build the Next.js admin app and copy the static output to `static/dashboard`. We'll cache dependencies to not slow down build too much. For the user-facing MyRoofGenius UI, we prefer deploying that separately on Vercel (or Netlify) since it's a public site. The CD pipeline can integrate with Vercel: either use Vercel's Git integration (so any push triggers Vercel build) or our GH Actions can do `vercel deploy` via token. Using Vercel's built-in might be simpler – just ensure environment is set there (like NEXT_PUBLIC_API_BASE if needed). The TemplateForge site (if planned similarly) would be another Vercel project. - **Database Migration:** We use Alembic for migrations [154]. The pipeline (or render start command) should run `alembic upgrade head` automatically so the Supabase/Postgres schema is up to date. Supabase might not allow remote DDL by default, but since we have the Service role key, we can run migrations via our app if configured. Possibly we might connect directly to Supabase's connection string to run Alembic. Alternatively, we maintain migrations for if someone uses self-hosted Postgres. In any case, apply new migrations as part of deploy. - **Cron Jobs / Scheduled Tasks:** If using Render, for tasks like daily plan we have choices: (1) Rely on our internal scheduler (the `scheduler.py` running inside app). (2) Use a separate cron job (Render Cron or GitHub Actions scheduled) to hit an endpoint or run a task. We plan to have an internal scheduler, but it might rely on the app staying awake. On Render free tier, app might spin down. But if we have constant usage or upgrade to

a plan, it's fine. Alternatively, configure a Render Cron job to GET our /agent/plan/daily at the scheduled time. We'll weigh that: For reliability, maybe schedule on Render: e.g. Sunday 16:00 hit `/agent/strategy/weekly` (or just rely on code). We'll incorporate whichever ensures tasks run even if no traffic. - **Secrets Management:** Our backend has endpoints for storing secrets at runtime (`/secrets/store`) [155]. But in CI/CD, we manage secrets through environment config in Render and Vercel. The `BRAINSTACK_API_KEY`, etc., should be set in Render's env for internal use. If new secrets are needed (e.g. NOTION_API_KEY if we implement, or PERPLEXITY cookie), those must be added to Render. We should document needed env vars so ops can set them. (The production readiness checklist doc can list these). - **Logging and Monitoring:** The deployed app will emit JSON logs to stdout (Render will capture those). We also send errors to Slack and Supabase logs table [78] [143]. So if something fails in production, the team gets alerted via Slack, and logs are in Supabase for further analysis (could view via the dashboard UI maybe with a /logs/errors endpoint [156] which we have). For metrics, we expose `/metrics` for Prometheus; if we set up a Prometheus/Grafana (maybe using Render's managed services or something), we could monitor. If not, we can rely on logs and Slack for now, given small scale. - **Auto-Documentation:** Claude is used to generate documentation (SOPs, etc.). We might integrate that into release process for things like updating this architecture doc or generating a changelog. For example, when merging, we could prompt Claude to update CHANGELOG.md based on PR titles. But that's optional polish. We do have a `CHANGELOG.md` already which could be manually or AI updated [157]. For now, we'll maintain it manually or via commit messages.

Deployment of Multi-Brand Setup: The backend is multi-tenant out of the box, serving multiple brands by data separation (project IDs, etc.). The frontends for each brand will be deployed separately (e.g. MyRoofGenius on its domain, TemplateForge on another). They all talk to the same backend API (which might be at api.brainstack.com or similar). We should ensure CORS allows those domains (settings can have `ALLOWED_ORIGINS`). We'll configure that (maybe allow all origins except lock down if needed). Also, rate limiting per IP is on (100/min by slowapi default [158] [80]), which should be fine for normal usage.

Rollbacks: If something goes wrong with a deployment, Render allows redeploying an older image. We should also keep the last stable image around. With GitHub Actions, we could push images with tags like `v1.0` etc. But since our deploy likely ties to main, we trust CI gating to catch issues. For frontends, Vercel automatically keeps previous deployments, easy to rollback with one click if needed.

Secrets (OpenAI, etc.) Rotation: If keys need rotation, it's an ops task, but our system has the `/secrets/store` to update them at runtime if needed (though presumably, restart is needed if the key is read at startup from env – unless we code to use DB-stored keys at runtime; could consider storing API keys in Supabase and pulling them dynamically so we can rotate without redeploying. But environment is okay for now).

Performance & Scale: The CI ensures tests, but for scale, we rely on Render auto-scaling (we can set concurrency for Uvicorn workers if needed, e.g. multiple workers or use Gunicorn). Celery tasks by default run in the same dyno on Render? If heavy tasks might need separate worker dyno or use RQ/BackgroundTasks for simplicity. The design uses Celery+Redis: we must deploy a Redis (Render provides a free Redis or use Upstash). Connect via `REDIS_URL`. Ensure to configure Celery in `celery_app.py`. Alternatively, given not extremely high load, FastAPI's async could handle some tasks inline for simplicity, but let's stick to Celery for robustness with long tasks and concurrency.

Claude/Codex Pipeline in CI/CD: The user asked for "*Claude file → test → deploy (Codex pipeline)*". This suggests perhaps a workflow where: 1. A markdown file (documentation or spec) is produced (maybe by Claude). 2. That goes through tests (maybe a human or automated check). 3. Then Codex generates code which is tested and deployed.

We are effectively doing that but mostly with humans writing code. If they want to integrate AI more: Possibly they envisage a future where writing a Markdown blueprint (like this doc or smaller feature spec) in the repo triggers an automated Codex job to create a branch with code changes. This is a bit advanced for fully automated CI. A safer approach is a CLI or GH Action that can be manually triggered to run Codex on a given spec file and open a PR with the changes. We can plan to include a script for that (but that might be beyond immediate scope). However, we can certainly embed clues in our CI for any .md file changes. For example, if `docs/blueprints/new_feature.md` is added, we could note in Slack that "This looks like a spec, consider running Codex generator." Or an action that picks up special commit messages to run certain tasks.

Given the question's phrasing, at minimum, we ensure our pipeline connects the pieces: - For now, the "Claude (for documentation/templates) and Codex (for code)" is a process done by developers using the blueprint. Our pipeline tests after the code is produced by Codex manually or semi-manually.

We will document in the README how to go from blueprint to code (maybe making a script to feed blueprint to OpenAI manually, or instruct usage of tools like GitHub Copilot with the blueprint).

Deployment Summary: - **Backend** on Render with Docker (with static admin UI baked in). - **User Frontend** on Vercel (MyRoofGenius domain) calling backend API. - Possibly **Admin Frontend** also accessible via Render (since static served) or via Vercel at a subdomain if we wanted. But internal likely fine at backend route (with login). - DB is Supabase cloud (we supply its URL and service key to backend). - Redis for Celery either Render or Upstash. - Slack alerts configured for errors. - Domain names configured: e.g. api.brainstackstudio.com CNAME to Render, myroofgenius.com to Vercel, etc. - Monitoring: Slack + maybe Sentry integration if we wanted for error tracing (we can integrate Sentry easily via their SDK for Python and React – not asked but worth noting as future improvement).

With this CI/CD and deployment setup, each code or prompt change goes through checks and gets deployed with minimal manual intervention, ensuring that the platform can evolve quickly and reliably. The **production readiness checklist** (docs/production_checklist.md) will be updated to include verifying CI passes, environment variables set, logging in Slack, etc., before a go-live [22].

All components described are designed to be modular (each part can be developed or updated independently), memory-aware (the system constantly logs and retrieves context to inform AI decisions), scalable (using cloud services and queues to handle load), and RAG-ready (the knowledge base integration ensures AI outputs remain grounded in our data). By following this blueprint, we set up a robust development and deployment cycle, where **Claude** helps with planning and documentation, and **Codex** accelerates coding, all validated by CI and deployed to cloud infrastructure seamlessly.

---

1 2 8 21 22 53 59 86 91 95 96 99 104 108 127 128 145 152 154 155 156 README.md
https://github.com/mwwoodworth/fastapi-operator-env/blob/7dbf0841f08b249d4e4fb833111827ad1f209824/README.md

3 103 README.md
https://github.com/mwwoodworth/fastapi-operator-env/blob/7dbf0841f08b249d4e4fb833111827ad1f209824/frontend/README.md

4 5 6 7 46 47 48 49 55 56 62 69 70 74 75 76 136 brainops_operator.py
https://github.com/mwwoodworth/fastapi-operator-env/blob/7dbf0841f08b249d4e4fb833111827ad1f209824/codex/brainops_operator.py

9 10 23 24 25 27 52 54 58 60 61 65 66 77 78 79 80 83 85 87 88 89 90 97 98 100 101 102 105 106 107 110 112 113 114 117 118 120 129 135 139 140 141 142 143 146 147 148 149 150 158 main.py
https://github.com/mwwoodworth/fastapi-operator-env/blob/7dbf0841f08b249d4e4fb833111827ad1f209824/main.py

11 dashboard.md
https://github.com/mwwoodworth/fastapi-operator-env/blob/7dbf0841f08b249d4e4fb833111827ad1f209824/docs/dashboard.md

12 13 30 37 92 115 116 memory_api.py
https://github.com/mwwoodworth/fastapi-operator-env/blob/7dbf0841f08b249d4e4fb833111827ad1f209824/codex/memory/memory_api.py

14 15 20 137 138 153 settings.py
https://github.com/mwwoodworth/fastapi-operator-env/blob/7dbf0841f08b249d4e4fb833111827ad1f209824/core/settings.py

16 26 ai_router.py
https://github.com/mwwoodworth/fastapi-operator-env/blob/7dbf0841f08b249d4e4fb833111827ad1f209824/utils/ai_router.py

17 57 backlog.md
https://github.com/mwwoodworth/fastapi-operator-env/blob/7dbf0841f08b249d4e4fb833111827ad1f209824/docs/backlog.md

18 19 31 32 40 44 45 50 51 memory_store.py
https://github.com/mwwoodworth/fastapi-operator-env/blob/7dbf0841f08b249d4e4fb833111827ad1f209824/codex/memory/memory_store.py

28 29 34 memory_utils.py
https://github.com/mwwoodworth/fastapi-operator-env/blob/7dbf0841f08b249d4e4fb833111827ad1f209824/codex/memory/memory_utils.py

33 41 42 43 gemini_memory_agent.py
https://github.com/mwwoodworth/fastapi-operator-env/blob/7dbf0841f08b249d4e4fb833111827ad1f209824/codex/tasks/gemini_memory_agent.py

35 36 doc_store.py
https://github.com/mwwoodworth/fastapi-operator-env/blob/7dbf0841f08b249d4e4fb833111827ad1f209824/codex/memory/doc_store.py

38 39 doc_indexer.py
https://github.com/mwwoodworth/fastapi-operator-env/blob/7dbf0841f08b249d4e4fb833111827ad1f209824/codex/memory/doc_indexer.py

63 64 67 68 82 119 130 agent_inbox.py
https://github.com/mwwoodworth/fastapi-operator-env/blob/7dbf0841f08b249d4e4fb833111827ad1f209824/codex/memory/agent_inbox.py

71 72 73 84 133 134 task_rescheduler.py

https://github.com/mwwoodworth/fastapi-operator-env/blob/7dbf0841f08b249d4e4fb833111827ad1f209824/codex/tasks/task_rescheduler.py

81 144 157 CHANGELOG.md

https://github.com/mwwoodworth/fastapi-operator-env/blob/7dbf0841f08b249d4e4fb833111827ad1f209824/CHANGELOG.md

93 94 131 132 recurring_task_engine.py

https://github.com/mwwoodworth/fastapi-operator-env/blob/7dbf0841f08b249d4e4fb833111827ad1f209824/codex/tasks/recurring_task_engine.py

109 151 page.tsx

https://github.com/mwwoodworth/fastapi-operator-env/blob/7dbf0841f08b249d4e4fb833111827ad1f209824/frontend/app/page.tsx

111 121 122 claude_prompt.py

https://github.com/mwwoodworth/fastapi-operator-env/blob/7dbf0841f08b249d4e4fb833111827ad1f209824/codex/tasks/claude_prompt.py

123 124 125 gemini_prompt.py

https://github.com/mwwoodworth/fastapi-operator-env/blob/7dbf0841f08b249d4e4fb833111827ad1f209824/codex/tasks/gemini_prompt.py

126 perplexityRelay.ts

https://github.com/mwwoodworth/fastapi-operator-env/blob/7dbf0841f08b249d4e4fb833111827ad1f209824/codex/research/perplexityRelay.ts