Project Taylor Microservices Architecture

Overview

Project Taylor is an automated resume tailoring and job application system. To achieve rapid development and future scalability, the system is designed as a collection of **microservices**, each responsible for a distinct part of the pipeline. This separation of concerns allows the solo founder to develop a Minimum Viable Product (MVP) quickly while laying the groundwork for a scalable SaaS product. The architecture leverages managed cloud services (MCPs) wherever possible – for example, using existing web scraping and AI APIs – to minimize custom code and accelerate developmentfile-bhni7m4babf28vykcjtusp. By integrating these services (for scraping, language processing, storage, etc.) as building blocks, the MVP can be delivered fast without sacrificing the ability to grow and scale later. Each component runs independently, enabling flexible scheduling (e.g. run the scraper during the day and the resume generator overnight) and independent scaling as needs evolve.

Microservices and Separation of Concerns

To support deployment via microservices, the system is broken into distinct modules, each possibly as a separate repository or deployable unit. The key microservices in Project Taylor include:

- **Job Scraper Service:** Responsible for acquiring job descriptions from external sources. Rather than building a scraper from scratch, this service can orchestrate an external scraping tool (e.g. an Apify actor) via API. For the MVP, an Apify actor performs the actual web scraping, and upon completion it triggers a webhook call to our systemfile-bhni7m4babf28vykcjtusp. The scraper service exposes an endpoint (e.g. `/webhook/new-job`) to receive job data in JSON. Its tasks are to validate and parse the incoming job description, then store it in the central database (Supabase) for further processing. This service can be scheduled to initiate scrapes periodically (e.g. daily) or respond to on-demand triggers. By isolating scraping, we can modify or scale it (e.g. scrape more sites or handle captchas) without affecting other components.

- **Filtering Service:** This service applies any filtering or ranking logic to incoming jobs. For example, it might compare the job description against the user's profile or preferences to decide if a job is relevant. It could filter out jobs that don’t match the target role or company criteria. In the pipeline, after a new job description is stored, the filtering service (running as a background worker or on a schedule) checks for new unprocessed jobs. Relevant jobs are marked for resume tailoring (e.g. updating a status field in the database) or placed onto a queue for the next stage. This separation ensures that only suitable jobs proceed, saving resources by not tailoring resumes for mismatched postings. The filtering logic can evolve (from simple keyword matching to more complex ML ranking) independently of other services.

- **Resume Generator Service:** This is the core module that generates a tailored resume for a given job description. It retrieves the relevant job info (and possibly the user's base resume and past templates) from storage, then uses AI and text processing to produce a customized resume draft. The generation involves multiple steps: extracting key skills and requirements from the job description (using an LLM API like OpenAI), finding matching experience or phrases in the user's profile (possibly using a vector similarity search via Supabase pgvector), and then assembling a new resume document. The service uses libraries like `python-docx` to programmatically manipulate the resume template (in .docx format) – inserting or modifying content to align with the jobfile-bhni7m4babf28vykcjtusp. Formatting is handled here as well, ensuring the output resume is polished. This service is compute-intensive and may be scheduled to run during off-peak hours (e.g. overnight) or scaled out to multiple instances if many resumes need generation concurrently. Because it runs independently, it can be updated (e.g. improving the AI prompts or formatting rules) without changing the scraper or email components.

- **Email Delivery Service:** Once a tailored resume is generated and stored, this service takes over to deliver the application. It pulls the finalized resume (from Supabase storage or the database) and sends it via email to the intended recipient. In an MVP for a solo job seeker, the email might be sent to the user's own inbox for review or directly to the job’s application email address. This service handles preparing the email content (possibly including a brief cover note), attaching the resume file, and using an email API or SMTP server to send it out. Using a transactional email service or SMTP credentials kept in secrets, it ensures reliable delivery. This service can also log the delivery status and update the database (e.g. marking the job as applied or emailing failed). By isolating email functionality, we can easily swap the email provider or add features (like CCing oneself, or integrating with job portals) later.

- **Orchestration & Common Services:** Rather than a single monolithic orchestrator, the system relies on event-driven orchestration using the database and possibly a task queue. The Supabase database acts as the integration point for these microservices: each service writes results (job data, filtering decision, resume path, send status) to the database. Services can detect new entries or status changes either via lightweight polling or database triggers. For example, the resume generator might poll for jobs marked “approved for tailoring” by the filter service. This decoupled approach means each microservice only needs to know *what data to consume or produce*, not the internal details of other services. Optionally, a simple message queue (e.g. using Redis or a library) could be introduced to pass events (like "job_ready_for_generation") to the resume service, improving real-time responsiveness. For the MVP, using the database for coordination is acceptable to reduce moving parts, but a task queue can be added for more robust scaling and retries. Common utility code (like data models or helper functions) can be shared in a lightweight internal package if needed, though keeping services as independent as possible is the goal.

Each of these microservices can be developed and deployed independently. This yields clear separation of concerns: the scraper deals only with job sourcing, the filter with decision logic, the generator with AI-powered document creation, and the email service with external communication. This modular design not only makes the system easier to understand and test, but also allows scaling or replacing each part without major impact on others. For instance, if a different AI service is to be used for generation, it impacts only the Resume Generator service. This design aligns with the philosophy of leveraging existing platforms for speed and focusing custom development on the glue between themfile-bhni7m4babf28vykcjtusp.

Repository Structure

Each microservice lives in its own codebase (repository or directory). Below is a high-level folder structure spanning all services, illustrating the separation and contents of each module:

graphql
CopyEdit

```
project-taylor/
├── job-scraper-service/
│   ├── Dockerfile
│   ├── README.md
│   ├── .env.example
│   ├── src/
│   │   ├── main.py           # Entry-point for the web server or
scraper runner
│   │   ├── scraper/          # Scraper logic (e.g. Apify API client
or scraping scripts)
│   │   └── api/              # API endpoints (e.g. FastAPI routes for
webhook)
│   ├── tests/
│   │   └── test_scraper.py   # Unit tests for scraping and webhook
handling
│   └── requirements.txt      # Python dependencies for this service
├── filter-service/
│   ├── Dockerfile
│   ├── README.md
│   ├── .env.example
│   ├── src/
│   │   ├── main.py           # Entry point, possibly a scheduler or
worker loop
│   │   └── filter_logic.py   # Filtering functions (criteria for job
relevance)
│   ├── tests/
│   │   └── test_filter.py    # Tests for filtering logic
│   └── requirements.txt
├── resume-generator-service/
│   ├── Dockerfile
│   ├── README.md
│   ├── .env.example
│   ├── src/
│   │   ├── main.py           # Entry point for the generation service
(could be API or worker)
│   │   ├── generator/        # Core logic for tailoring resumes
│   │   │   ├── tailor.py     # Functions orchestrating the tailoring
process
│   │   │   ├── llm_client.py # Utility to call LLM APIs (e.g. OpenAI)
│   │   │   ├── template.docx # Resume template(s) for formatting
│   │   │   └── formatter.py  # Code to apply formatting using
python-docx
│   │   └── data/             # e.g. sample resumes, reference skills,
etc.
│   ├── tests/
│   │   ├── test_tailor.py    # Tests for resume generation logic
│   │   └── test_formatter.py # Tests for formatting output
│   └── requirements.txt
├── email-delivery-service/
│   ├── Dockerfile
│   ├── README.md
│   ├── .env.example
│   ├── src/
│   │   ├── main.py          # Entry point for the email sender (could
be a small script or API)
│   │   └── emailer.py       # Functions to compose and send emails
(using SMTP or API)
│   ├── tests/
│   │   └── test_email.py   # Tests for email sending logic
│   └── requirements.txt
├── common-utils/             # (Optional) shared utilities or schemas
│   ├── logging.py          # Common logging setup for consistency
│   ├── db.py               # Database helper (for connecting to
Supabase/Postgres)
│   └── ...                 # Other shared code, if any (could also be
a pip package)
└── docker-compose.yml        # Compose file to run all services
together (for dev/testing)
```

Repository Highlights:

- Each service directory (e.g. `job-scraper-service/`) contains everything related to that microservice. This includes a `Dockerfile` for containerizing the service, a `README.md` with usage/deployment instructions, an example environment file (`.env.example`) listing required environment variables (secrets or configs), the source code under `src/`, and a `tests/` folder with unit tests. Each also has its own `requirements.txt` (if Python) or equivalent dependency file, so services remain independent in dependencies.

- **Source Code (`src/`):** The source folder is organized by feature for that service. For instance, the scraper service has an `api/` subfolder for endpoint definitions (if using FastAPI to accept webhooks) and a `scraper/` subfolder for the logic to initiate or coordinate scraping jobs. The resume generator service has submodules for the generation and formatting logic – separating concerns like calling the LLM API (`llm_client.py`) from applying the template formatting (`formatter.py`). This keeps code modular and testable. The resume service may also include static template files (like a base `.docx` template) or sample data under its directory.

- **Dockerfile:** Each microservice is packaged as a container. The Dockerfile will define how to build the service (e.g. starting from a Python image, copying the code, installing `requirements.txt`, and setting the command to run `src/main.py`). Containerizing each service ensures environment consistency across development, testing, and production, and makes it easy to deploy on any platform supporting Docker.

- **Environment Variables:** The `.env.example` files document what configuration each service needs (for example, the scraper might need `APIFY_API_TOKEN` if it needs to call Apify, the resume service might need `OPENAI_API_KEY`, and all might need `SUPABASE_URL` and `SUPABASE_SERVICE_KEY` to access the common database/storage). We **do not** store actual secrets in these files – real secrets are injected via environment (more on that in a later section). This setup follows the twelve-factor app principle of separating config from code.

- **Tests:** Each service's tests focus on that service's functionality. For example, `test_filter.py` would feed sample job postings into the filtering logic to verify that the criteria are correctly applied. The resume generation tests might include checking that given a known job input and a dummy profile, the output document contains certain expected sections or keywords. By isolating tests per service, we can rapidly catch bugs within a module. Later, end-to-end tests can be added to cover the entire pipeline.

- **Common Utilities:** In a microservices architecture, it's often best to avoid tight coupling, so sharing code is optional. However, to avoid repeating boilerplate (like setting up a database connection or logging format in each service), a small `common-utils` library can be created. This could be a separate package that each service installs, or simply a directory included via git submodule or copied. It might contain things like a standardized logging setup (so all services log in the same format) and database helpers to connect to Supabase. This ensures consistency without duplicating code, but we remain cautious: changes in common code affect all services, so this should be used sparingly and kept very stable.

- **Docker Compose:** The root `docker-compose.yml` is provided for convenience in development and testing. It can define all services (scraper, filter, resume, email) and any supporting components (for example, a local Postgres container if we simulate Supabase locally, or perhaps a Redis container if we use it for a task queue). With one command, a developer can bring up the entire stack on their machine, which is useful for integration testing. This also serves as documentation of how services interact (e.g. linking them on a virtual network and setting environment variables so they can reach the database). The compose file can mount code directories for rapid iteration. For production deployment on a service like Railway, the compose is not directly used, but it ensures we can reproduce the environment anywhere – fulfilling the **Infrastructure as Code (IaC)** principle for reproducibility.

Overall, this repository structure keeps each microservice self-contained. In practice, the solo founder could choose to keep all these folders in a single repository (monorepo style) for simplicity, or separate them into multiple repos (one per service). A monorepo makes it easier to coordinate changes across services and track issues in one place, whereas separate repos might be preferred if different components are developed on different timelines or open-sourced individually. In either case, each service can be built, tested, and deployed independently.

Managed Service Integration (MCP-as-a-Service)

A key design choice is to integrate **Managed Cloud Platforms (MCPs)** and third-party services for heavy lifting in web scraping, AI processing, storage, and email delivery, rather than reinventing those. This *"MCP-as-a-Service"* integration means each microservice often acts as an orchestrator or glue to these external capabilities:

- **Web Scraping (Apify or Similar):** The Job Scraper service leverages Apify (a cloud scraping service) instead of containing complex scraping logic itself. The service's role is to trigger the Apify actor (if not scheduled externally) and consume the results. In practice, the founder's existing Apify actor can be scheduled on Apify's platform to run searches on job boards, then call our webhook with the datafile-bhni7m4babf28vykcjtusp. This setup decouples scraping from our infrastructure – Apify (or another MCP) handles browser automation, proxies, and site-specific parsing, while our system simply reacts to the final data. If Apify is not available, the scraper service could fall back to a simpler built-in scraper or use another API (like an RSS feed or LinkedIn job API if available). For future scalability, we can swap out Apify with an in-house scraping cluster if needed, but for the MVP, using it *as a service* saves a lot of development time.

- **LLM Generation (OpenAI API):** The Resume Generator service relies on an external Large Language Model (e.g. OpenAI's GPT-4 API) to perform tasks like extracting key skills from the job description or refining bullet points. This approach of using an API call to an AI service means we don't need to host a model ourselves. It dramatically simplifies the MVP, as we leverage a state-of-the-art NLP model with minimal integration codefile-bhni7m4babf28vykcjtusp. The generator service prepares prompts and calls the API (using the secret API key), then handles the response (parsing out suggested text or analysis). By treating OpenAI (or a similar LLM provider) as a service, we can get high-quality results quickly. The system is designed such that if costs or other factors become an issue, this component could be replaced with a smaller local model or a different provider later, without affecting other services. For now, the priority is fast, reliable output using a proven AI service.

- **Document Formatting (Python Libraries):** While formatting is done within our resume generator microservice, we lean on open-source libraries (an example of leveraging existing software). The `python-docx` library is used to manipulate Word documentsfile-bhni7m4babf28vykcjtusp, and possibly other libraries for PDF conversion if needed. These libraries are effectively "services" in that they provide well-tested functionality we integrate. By using them, we avoid writing low-level file handling code. If needed, one could also use cloud APIs for document conversion (like an API to convert HTML to PDF), but in this case `python-docx` suffices and keeps data local to the service.

- **Storage (Supabase):** For storing both data and files, the architecture uses Supabase, a managed backend service built on PostgreSQL. Supabase provides a database for structured data (job postings, user info, logs, etc.) and an object storage for files (like storing the generated resume file). Instead of running our own database or file server, we treat Supabase as an external data service. Each microservice that needs to read or write data uses Supabase's RESTful API or client libraries (with credentials) to do so. For example, when the scraper service receives a job, it uses a Supabase client to insert a record in a `jobs` table. The resume service might fetch templates or past resume content from a `resumes` table or a storage bucket. Supabase's **pgvector** extension is used by the resume service to perform similarity search on text embeddings (helping match resume content to the job description)file-bhni7m4babf28vykcjtusp – this means we don't need a separate vector database. By centralizing state in a managed Postgres, all services have a single source of truth and we avoid the complexity of syncing data across services. Supabase also handles user authentication (not heavily used in an MVP with one user, but crucial for a future multi-user SaaS) and offers easy scalability for the database. Treating it as an MCP means configuration and scaling of the DB is largely handled by Supabase, not by our codefile-bhni7m4babf28vykcjtusp.

- **Email/Delivery (SMTP or Email API):** The Email Delivery service could use a simple SMTP server (e.g. Gmail's SMTP or another provider) by storing credentials as secrets, or it could integrate with an email sending service like SendGrid, Mailgun, or AWS SES. Using such a service means we don't host an email server or worry about deliverability issues – instead, we send a request to the email API with the message and it handles the rest. For the MVP, an easy route is to use something like Gmail SMTP (with an app password) since it's quick to set up for a personal project. As the system scales or if it becomes a product, a more robust email API service would be used to handle higher volume and provide analytics. In either case, the email service module just needs to know how to talk to the external email system (via SMTP or REST API) and the credentials to do so. This again follows the pattern of offloading complexity to an external platform so we can focus on core logic.

- **Other Integrations:** The architecture is open to integrating additional services like Zapier or n8n for workflow automation. In the MVP design, after emailing the tailored resume, one could use Zapier to log the application in a Google Sheet or create a Trello card for tracking. These are ancillary, but they fit the same philosophy: use managed tools for supporting workflows rather than coding everything. The microservices would simply emit events or call a webhook that Zapier/n8n listens to, if such features are addedfile-bhni7m4babf28vykcjtusp.

By using these MCP services, we ensure that each microservice in Project Taylor remains relatively small and focused on orchestration and light processing, rather than doing heavy-duty computing itself. This drastically reduces development time and bugs – for instance, we rely on OpenAI for NLP correctness and Apify for scraping reliability. It **leverages powerful existing platforms to accelerate the MVP**file-bhni7m4babf28vykcjtusp. Importantly, each integration is designed to be swappable: if a service becomes a bottleneck or too costly, we can replace that MCP with another solution by only modifying the respective microservice. This strategy provides a fast path to a working system and the flexibility to evolve into a more proprietary solution if needed later.

Deployment and Scalability

The microservices architecture allows flexible deployment strategies. For the MVP, we target a cloud deployment so that the system can run continuously and scale on demand, but we also ensure the environment can be reproduced locally or in different cloud providers via containerization and Infrastructure as Code.

- **Containerized Deployment (Docker):** Each microservice is packaged as a Docker container (using the Dockerfiles mentioned earlier). This guarantees that the code runs in a consistent environment (with specific OS libraries, Python versions, etc.) everywhere. For cloud deployment, we can use a platform like **Railway** (a popular modern PaaS) which directly supports deploying from Dockerfiles. Railway was chosen in the planning due to its developer-friendly usage and support for scaling and schedulingfile-bhni7m4babf28vykcjtusp. On Railway, each microservice would be defined as a service (either in one project or separate projects). We can push our code to GitHub and have Railway auto-build and deploy the container for each service. Because each service is separate, we can deploy them independently – for example, update the resume generator without redeploying the scraper – enabling faster iterations.

- **Independent Scaling:** With microservices, each component can scale based on its specific load. Railway (and similar platforms) allows adjusting resources or number of instances per service. For instance, if generating resumes becomes a CPU-intensive task that is slow, we could allocate more RAM/CPU to the Resume Generator service or run multiple instances of it. The scraper service might only need one instance running occasionally, whereas the resume generator could scale to N workers processing jobs in parallel. Similarly, the email service might be light but could be scaled out if sending many emails concurrently or if we integrate with many job application endpoints. Since the services communicate via the database or messages (not directly), horizontal scaling (adding instances) doesn't require changes in code – e.g., multiple generator instances can all pull from the job queue table and work in parallel. This architecture is cloud-ready and can handle increased throughput by tuning each microservice's deployment size or count.

- **Scheduling and Orchestration:** Some components do not need to run 24/7 at full throttle. For example, the Filter service might only need to wake up when new jobs arrive, or periodically scan the DB. The system can use scheduling to optimize operations:

    - **Cron Jobs:** Railway supports a cron feature to hit endpoints on a schedule or run one-off tasks. We could set up a cron job to trigger the Apify scraper actor every morning at 9 AM, for instance, or to call an endpoint on the resume service to process any pending jobs at midnight. This way, even if the services are always running, they perform heavy work only at planned times. The Resume Generator could also simply run continuously but check the DB and sleep if no jobs are queued, which is another approach.

  - **Event-Driven Triggers:** Supabase can emit database triggers or webhooks when new data is inserted. We might leverage that: e.g., when a new job entry is added, a database trigger could call a serverless function or notify the resume service. However, a simpler approach for MVP is polling or scheduled checks due to simplicity. In any case, the architecture supports decoupling: one service can schedule another or data changes can implicitly trigger next steps.

  - **Task Queue:** For more robust orchestration, a message queue system (like Redis or RabbitMQ) can be introduced. We could have the Scraper service publish a message “job X ready” to a queue, which the Resume Generator service listens to. This is a common scalable pattern ensuring real-time processing and letting the queue handle load buffering. While perhaps not implemented on day one (to reduce complexity), the code is structured to allow adding such a queue later without overhaul. Using a queue also facilitates retries and monitoring of tasks (e.g., requeue a failed job), contributing to reliability.

- **Local vs Cloud Execution:** For development and testing, the entire system can run locally using Docker Compose, connecting to a local development database. This is useful for debugging and ensures that the code doesn’t rely on anything that only exists in production. However, for the actual job search use-case, running in the cloud is ideal. Cloud deployment (on Railway/Supabase) ensures the pipeline keeps running even if the founder’s laptop is off, and it benefits from stable internet and uptime. The **MCP servers** in this context refer to the cloud servers running these containers on platforms like Railway and the Supabase cloud services. We recommend running all microservices in the cloud for the live system, given the need for continuous operation (free-tier services that pause after inactivity are problematic for automationfile-bhni7m4babf28vykcjtusp). That said, the design allows flexibility. If needed, certain heavy components could be run on a local machine or different environment. For example, if one wanted to use a local GPU for AI, the resume generator could be run on a local workstation (pointing to the same database) overnight. This wouldn’t require changes in other services – they would still put jobs in the queue/DB and expect results. In summary, **cloud is preferred** for its managed scaling and reliability, but the containerized microservice approach means components can be deployed to any mix of local or cloud environments as long as they can network together.

- **Infrastructure as Code (IaC):** To support reproducible environments, we treat infrastructure setup as code where possible. The Docker configuration is one form of IaC. Additionally, we can use tools like **Pulumi or Terraform** to script the provisioning of our services. For instance, Pulumi (in Python or TS) could be used to create a Railway project, set up each microservice (pointing to the Docker image or repo), configure environment variables (secrets), and create a Supabase instance and the necessary database schemas. This way, the entire stack (compute and data services) can be spun up in a new environment or updated consistently. While setting up Pulumi for Railway/Supabase might be optional for an MVP, it becomes valuable as the project grows or if deploying to other cloud providers. Even using **Railway's CLI** with a config file or **Supabase CLI** for migrations can be part of an IaC approach. The goal is to avoid manual clicks for setup – instead, check in configuration scripts or YAML definitions so that the deployment process is repeatable and version-controlled. This approach ensures that if the founder needs to migrate to a new account or recover the infrastructure, they can do so quickly, and it sets the stage for more formal DevOps if the project becomes a SaaS with multiple environments (dev/staging/prod).

In sum, the deployment architecture is designed for **fast MVP launch and easy scaling**. Using Docker and platforms like Railway means minimal server maintenance and the ability to adjust resources per service quickly. The microservices can run independently, scheduled intelligently to optimize resource use, and scaled horizontally as usage grows. This approach can seamlessly transition from a scrappy MVP to a more robust production deployment by gradually increasing resources or introducing more sophisticated orchestration (like Kubernetes or serverless functions for each task) when needed – but none of that is required on day one.

Security and Secrets Management

Handling credentials and sensitive data securely is paramount, even for a small MVP. The architecture incorporates best practices for secrets management:

- **Environment Variables:** All secrets (API keys, database URLs, service credentials) are kept out of the code and config files, and are provided to the services through environment variables. Each service's `.env.example` file documents the keys needed, but actual values are set in the deployment environment. For example, the Scraper service will expect something like `APIFY_API_TOKEN` in its environment, the Resume service expects `OPENAI_API_KEY`, and so on. This follows the principle of not hard-coding secrets and allows different values in dev, test, and production. On a platform like Railway, we can use their integrated secrets management to set these env vars in the project settings. Railway encrypts these values and makes them available to the container at runtime. Supabase provides a secure connection string for the database which we also store as an env var (often as `DATABASE_URL` or separate host, password, etc.). By using environment variables, we ensure that if the code is shared or pushed to GitHub, no sensitive information is exposed.

- **Secret Storage and Access:** In local development, developers can use a `.env` file that is loaded by a tool (like Python's `python-dotenv` or Docker Compose's env_file feature) – this file would be in the `.gitignore` so it never goes to source control. In production, secret values are directly configured in the hosting platform's dashboard or via IaC scripts (e.g., Pulumi can pull from a secure source or prompt the user for them, then set them in the target environment). For an MVP with a solo developer, manually configuring these a single time is manageable. As the project grows, one might integrate a dedicated secrets manager (such as HashiCorp Vault, AWS Secrets Manager, etc.), but that may be overkill initially. The important practice is that secrets are *centralized and changeable* without code changes – if a key gets compromised or needs rotation, we update the environment, not the code.

- **Credential Scope:** Each service only gets the credentials it needs. For example, the Email service doesn't need the OpenAI API key, and the Resume service doesn't need the SMTP password. We follow the principle of least privilege by only injecting the relevant secrets into each microservice's environment. This limits the blast radius if one service were somehow compromised. The Supabase service role key (which allows read/write to the database) might be needed by multiple services; to handle this safely, Supabase can issue different API keys or we might use row-level security with an authenticated service user if needed. For MVP, using the service role key in each service (and keeping it secret) is acceptable. We just ensure it's not exposed in logs or error messages.

- **Secure Communication:** We ensure all communication with external services is over HTTPS or secure channels – for instance, the webhook from Apify should be an HTTPS endpoint (Railway provides HTTPS endpoints by default). Similarly, connections to Supabase use SSL. No sensitive data is transmitted in the clear. Within our system, if using a queue or internal APIs, we might not need encryption (assuming everything runs within a private network on the cloud), but it's good to be mindful if any service endpoints were exposed publicly (they should be protected or secret).

- **API Keys and Permissions:** For external APIs like OpenAI or Apify, we treat those keys carefully. They are stored as env vars (e.g., `OPENAI_API_KEY`) and not logged. If our code calls these external services, we handle errors gracefully without printing the key. We also monitor usage to ensure keys aren't abused. In a future multi-user scenario, we'd have to secure user data as well, but in this single-user MVP, the main secrets are service credentials.

- **Database Security:** Supabase is a managed Postgres with its own authentication. We will use the service key in our server-side services to read/write data. Supabase can restrict access to data through its policies. We likely configure the database so that it's not open to the world except via the Supabase API (or direct connection using the key). This means our microservices should talk to Supabase through the provided URL and API, rather than an insecure connection. Supabase also handles user authentication tokens if needed (for example, if later adding a user-facing app), which is out of scope for now. Regular backups are automatically handled on Supabase Pro tierfile-bhni7m4babf28vykcjtusp, but we should export critical data periodically if on free tier.

In summary, **secret management relies on environment-level configuration and trusted platform features**. By following these practices, the system avoids common pitfalls like committing API keys to the repo. Even as a fast-built MVP, it adopts these precautions so that scaling up and adding more collaborators or moving to production will not require a painful security retrofit. It's easier to do it right from the start: use env vars for secrets[learn.microsoft.com](learn.microsoft.com), store them securely, and give each service only what it needs.

Logging, Monitoring, and Resilience

Robust logging and the ability to recover from failures are vital for an automated system. Project Taylor's architecture includes plans for thorough logging, error handling with retries, and basic monitoring:

- **Structured Logging:** All microservices will emit logs that include structured information, making it easier to debug and trace the pipeline. Rather than ad-hoc print statements, we use a logging framework in each service (Python's `logging` module configured to JSON or key-value output). We include context in log messages, such as a job ID or job title, so that events from different services can be correlated. For example, when a job is scraped, the Scraper service logs "Job X received" with X being some identifier; the Resume service later logs "Generating resume for Job X" and the Email service "Sent resume for Job X". By searching the logs for "Job X", we can reconstruct the sequence across services. Railway aggregates logs from each service in its console, which is convenient for initial monitoringfile-bhni7m4babf28vykcjtusp. In production, we might forward logs to a central system (like LogDNA, ELK stack, etc.), but initially, the platform logs and console output suffice. We plan the log format early to keep it consistent as complexity grows.

- **Monitoring and Alerts:** In the MVP stage, active monitoring can be simple: regularly check the logs and ensure jobs are flowing. Railway provides basic metrics (CPU, memory usage of containers) and Supabase provides database metricsfile-bhni7m4babf28vykcjtusp. If something crashes, Railway can be configured to restart the service automatically. We should set up alerts for critical failures – for instance, if the resume generator throws an exception or if an API call fails repeatedly, it should log an error that is noticeable. As a solo developer, simply having error logs may be enough (one can check daily), but to optimize, one could integrate a service like Sentry for exception tracking. This would catch exceptions in any microservice and send an alert/email when something goes wrong. Additionally, uptime monitoring (like a ping to the webhook endpoint or a health-check) can be used to ensure the services are running. Supabase can also trigger alerts if database usage spikes unexpectedly. These measures ensure that if something breaks (e.g., the Apify webhook stops coming, or OpenAI API quota is hit), the founder can react quickly, which is crucial when running a job application pipeline.

- **Retries and Error Handling:** Each microservice implements robust error handling around external calls. For example, if the Scraper service's webhook processing fails to write to the database (say, due to a network glitch), it should catch that and retry or at least queue the data to try again. The Resume Generator, especially, deals with external APIs (OpenAI) which can time out or return errors. We incorporate a **retry mechanism with exponential backoff** for calling these APIsfile-bhni7m4babf28vykcjtusp. This means if the first attempt fails, wait a short period and try again, increasing the wait time for subsequent retries. This helps handle transient issues (like a momentary network issue or rate limit). We might use a library (like Tenacity for Python) to simplify implementing backoff. If after several retries the call still fails, the service should log a clear error and move on rather than hang indefinitely. In such a case, the job can be marked as failed or left for manual review. Similarly, if the Email service fails to send (SMTP server down, etc.), it should retry sending a few times before giving up and logging an alert.

- **Idempotency and Task Queues:** When dealing with retries or restarts, it's important that tasks are not duplicated or corrupted by partial execution. We design the process to be **idempotent** where possible. For example, if the resume generation for Job X fails halfway, we can safely retry it because generating the resume from the same inputs should yield the same result (and if a partial file was saved, the service can overwrite it). Using the database to track status helps: a job might have a status field ("pending", "in_progress", "completed", "failed"). The Resume service would set "in_progress" when it starts, and "completed" when done. If a crash happens, the job might remain "in_progress" without completion – a cleanup routine or manual trigger could reset it to "pending" for retry after a timeout. Alternatively, a task queue system could automatically not acknowledge the message if the worker dies, and another worker would retry the same task. For MVP, a simpler approach is acceptable: even a daily script could look for jobs not completed and retry them.

- **Transactional Integrity:** All writes to the database should ideally be done in a way that preserves integrity. Supabase/Postgres allows transactions – e.g., inserting a new resume and updating job status together. We should utilize that for critical sections, or do updates in the correct order to avoid inconsistent states. Logging each step (start and end of each service's handling of a job) provides an audit trail to debug issues.

- **Testing Failure Scenarios:** As part of development, we will test how the system behaves under failure. For instance, simulate an OpenAI API failure in a test to verify our retry logic actually retries and then gives up gracefully without crashing the service. Similarly, unplug the network in a test environment to see if the email service queues the email for later. These tests ensure our resilience mechanisms work as intended.

By implementing these logging and resilience practices, the system gains reliability which is crucial for an automated pipeline running largely unattended. We want the pipeline to be able to run for days continuously, applying to jobs, without constant oversight – and for the founder to be confident that if something goes wrong, it will either self-correct (via retries) or at least report the issue clearly. This focus on robustness from the outset was a recommended practice in the technical planfile-bhni7m4babf28vykcjtusp, and it will pay off by reducing downtime and missed opportunities. As volume grows, we can enhance monitoring (e.g., integrate with a full APM solution, use a dashboard to track how many jobs processed, etc.), but the MVP will have the essentials: good logs, some alerts, and the ability to recover from common transient failures.

Testing and Continuous Integration

Given the ambitious goal (fully automated job applications) on a short timeline, testing is critical to ensure each part works correctly. The plan is to test at multiple levels and use continuous integration (CI) to catch issues early:

- **Unit and Module Testing:** Each microservice has its own test suite focusing on its logic in isolation. For example, in the Resume Generator service, we test the function that calls the LLM by mocking the OpenAI API – ensuring that our prompt formatting and response handling work as expected (without actually calling the API). We test the template formatting by providing sample data and verifying the output document has the right sections. In the Filter service, we can test the criteria function with various job inputs (some that should pass, some that should fail) to make sure the logic is correct. These tests run quickly and give confidence that the building blocks behave properly. We also include edge cases (like empty fields in a job description, or extremely long descriptions) to ensure robustness.

- **Integration Testing:** After verifying individual services, we perform integration tests for the interactions. One approach is to use the **docker-compose setup in a CI environment** to launch all services in test mode and simulate a full pipeline run. For example, a test could insert a dummy job into the database (bypassing the actual scraper), then run the filter logic (perhaps triggered manually in the test), then run the resume generator, etc., ultimately checking that a resume file was created and an "email sent" log entry appears. This can be facilitated by providing test configurations or flags in services (like a mode where the email service doesn't actually send an email but just logs the action). Another integration test scenario: simulate the webhook – call the Scraper service's webhook endpoint with a sample job JSON and then observe if the database now has a new job entry and eventually a resume entry. These tests ensure that the services can work together using the real database and that our data schemas align. We might use a separate testing schema or a SQLite in-memory database for integration tests to avoid messing up real data.

- **End-to-End Dry Runs:** Before deploying or as a final verification, we run a full end-to-end test with all pieces in place (perhaps in a staging environment or using the dev environment but with real services). This might involve running the Apify actor on a known small search, letting it trigger the webhook, and seeing a resume get generated and emailed to a test email address. This final sanity check covers the entire pipeline in a real-world scenario. Given the one-month timeline, we want to catch any integration issues early so the actual job applications go out flawlessly.

- **Continuous Integration Pipeline:** We set up a CI pipeline (for instance, using GitHub Actions or GitLab CI depending on where the repos are) to automate the testing processfile-bhni7m4babf28vykcjtusp. Every time we push changes to a microservice, the pipeline will run that service's tests. For changes that affect multiple services (if in one repo), it can run all tests. The pipeline can also build the Docker images to ensure the Dockerfile is correct. This automated testing guardrail is important as we rapidly iterate – it will catch if a change in code breaks something that was previously working (regressions). Additionally, the CI pipeline can run linting and security scans (like checking for vulnerable packages) to maintain code quality.

- **Continuous Deployment (CD):** For fast MVP development, we might opt for auto-deployment on passing tests. Railway integrates with Git repositories so that on each push to main (or a specific branch), it can rebuild and deploy the service. We'll likely use that feature: once tests pass in CI, we allow the new version of the service to deploy. This means we can push fixes or improvements multiple times a day and always have the latest running. We will do this carefully—perhaps using a dev environment vs production environment if needed. But since initially the user base is just the founder, continuous deployment to production is fine as long as we have confidence in our tests.

- **Testing Infrastructure as Code:** If we use Pulumi or similar, we will also version-control our infrastructure definitions and possibly test them (at least manually verify that a fresh deploy works). For example, we might have a Pulumi script to set up a Supabase schema (tables for jobs, resumes, etc. with appropriate columns). We'd test that by running it on a test project and then running the services against that to ensure all columns and permissions are correct.

- **User Acceptance and Iteration:** Finally, since the founder is the end-user, they will effectively test the system in real usage – applying to some initial jobs and observing results. Feedback from this manual testing will be quickly folded back into development. Perhaps the filtering needs tweaking or the resume output needs adjustments; the microservice architecture makes it easier to pinpoint where to make the change (e.g. filtering criteria service or template content in resume service) without unintentionally breaking other parts.

This testing strategy (unit -> integration -> end-to-end) aligns with the recommendation to test components in isolation, then in concert. It ensures reliability which is especially important given the time-sensitive goal (securing a job interview via this system). Moreover, establishing CI/CD now sets up good habits for the future – if the project becomes a full SaaS with multiple users, these automated tests and deployments will support continuous improvement without constant manual checking.

Rationale and Future Outlook

The proposed architecture is designed with **both immediate needs and future expansion** in mind. Here we summarize why these choices make sense for the solo founder's one-month goal and how they pave the way for turning Project Taylor into a scalable product:

- **Fast Development:** By using microservices that integrate managed services, the founder maximizes development speed. Each piece is simplified by leveraging existing technology (e.g., using Apify instead of writing crawlers, calling an AI API instead of training a model)file-bhni7m4babf28vykcjtusp. The glue code in each microservice is straightforward, meaning the MVP can be up and running quickly. This addresses the urgent need to start applying to jobs within weeks. The core pipeline – from job description to emailed resume – is prioritized and built with minimal overhead.

- **Isolation of Complexity:** Each microservice addresses one aspect of the problem, which reduces the cognitive load. The founder can work on one component at a time without breaking others. For instance, if initial results show the resume formatting is weak, almost all changes will occur in the Resume Generator service alone. This isolation also means bugs are easier to trace (logging per service helps pinpoint which stage failed). It aligns with the Unix philosophy of small, single-purpose components, and with the concept of separation of concerns crucial for maintainable systems.

- **Scalability and SaaS-readiness:** Although the MVP might only serve one user (the founder), the architecture naturally extends to multiple users and higher volumes. Adding new users could be as simple as adding new records in the database and perhaps scaling out the services. Because the state (jobs, resumes) is in a central database, microservices remain stateless and easy to replicate for scale. If Project Taylor becomes popular, the founder can deploy more scraper instances to cover more job boards, more generator instances to handle more resumes simultaneously, etc., without redesigning the system. Also, new features can be added as new services or extensions of existing ones. For example, a "Cover Letter Generator" microservice could be introduced later without disrupting the resume flow – it would take the job and perhaps the generated resume as input and produce a tailored cover letter, then the Email service could attach it. The current design is flexible enough to accommodate such growth.

- **Technology Stack Justification:** The combination of Railway and Supabase was confirmed to be a strong foundation for this projectfile-bhni7m4babf28vykcjtuspfile-bhni7m4babf28vykcjtusp. Railway provides an easy way to deploy these microservices (which are Dockerized Python FastAPI or worker apps) and manage them (with logging, cron, and environment config) with minimal DevOps effort. Supabase provides the data layer (Postgres DB, storage, authentication, and even vector search) out-of-the-box, which means we didn't need to set up a separate database server or ElasticSearch for similarity. This reduces setup time and operational burden. The use of Python and its rich ecosystem (FastAPI, `python-docx`, machine learning libraries) is ideal for an AI-driven project and is something the founder can iterate on quicklyfile-bhni7m4babf28vykcjtusp. All these choices ensure the MVP is not bogged down by infrastructure work and can focus on functionality.

- **Security and Reliability from Day 1:** Even as an MVP, we adopted proper secrets handling, logging, and error recovery strategies. This is slightly more upfront work but saves time later – we won't have to scramble to add error handling after missing out on an application due to a silent failure, for example. The recommendation to include retries and structured logging earlyfile-bhni7m4babf28vykcjtusp was heeded. This means the system is more trustworthy; the founder can let it run with more confidence, and if something does go wrong, it will be easier to diagnose and fix. For a SaaS future, these aspects (security, reliability) are non-negotiable, so we've essentially built a prototype that isn't just a happy-path demo, but a solid foundation that can be extended.

- **Rationalizing Microservices for an MVP:** One might question if microservices are overkill for a one-person MVP. In this design, the microservices approach is justified by the nature of the problem – distinct tasks that naturally fit separation. It also allowed using different third-party services cleanly for each part. The overhead of managing multiple services is mitigated by using the same tech stack for each (mostly Python) and a platform like Railway that handles much of the service management uniformly. The benefit is that if any one part of the pipeline needs to be changed or replaced (which is likely in an evolving project), it can be done so with minimal impact on the rest. For example, if Apify's free tier limits become an issue, the Scraper service could be modified to use a different method or an alternate API without needing to touch resume or email code. This modularity de-risks the project – we have contingency paths for each component.

- **Supporting Future Growth:** When turning Project Taylor into a full SaaS offering, some enhancements would be needed (user management, UI, multi-tenancy in the database, more robust task distribution), but the architecture won't require a complete rewrite. We can introduce an API Gateway or a front-end that talks to these services; each user could have their own set of records in the same database (Supabase can handle row-level security for multi-user separation). The microservices could be deployed in a Kubernetes cluster or scaled via serverless functions for even greater scale. Essentially, the MVP's microservices become the building blocks of a larger system. The code written now will largely carry forward, which is an efficient use of the short development time available.

In conclusion, the microservices-based architecture for Project Taylor provides a clear, organized approach to building the automated resume tailoring system. It balances **speed** (through leveraging MCP services and simple, focused modules) with **future-proofing** (through scalability, modularity, and good practices). The design choices made – from the folder structure to the use of cloud platforms – all serve the dual purpose of getting a working product quickly and setting up for a potential SaaS that can help many job seekers. This comprehensive yet agile architecture will help the solo founder not only build the system within a month but also adapt and grow it in the months to come, as new opportunities and requirements emerge.

Sources: The design principles and technology choices are informed by the MVP architecture report and best practices, which emphasize using managed platforms for faster developmentfile-bhni7m4babf28vykcjtuspfile-bhni7m4babf28vykcjtusp, separating compute and state between Railway and Supabasefile-bhni7m4babf28vykcjtusp, and implementing robust error handling and logging from the startfile-bhni7m4babf28vykcjtusp. These choices ensure that Project Taylor's implementation is both rapid and robust, aligning with the goal of quickly securing a job while laying a foundation for a scalable product.