ChatGPT

GoGetScholarship – Technical Architecture Report

Executive Summary

Problem: Many students struggle to find and apply to scholarships that fit their profile. GoGetScholarship addresses this by intelligently matching students to scholarship opportunities and even drafting personalized application essays. The system combines *vector similarity search* with *LLM (Large Language Model)* reasoning to rank scholarships by best fit, and uses AI to generate tailored essay drafts for each scholarship.

Solution Approach: The platform ingests scholarship descriptions into a *vector database* and uses **Voyage 3.x** embedding models to numerically encode (embed) both scholarship text and student profiles into a high-dimensional space. Given a student's profile, it first performs a **semantic vector search** in the database to retrieve relevant scholarships. These candidates are then **re-ranked by an LLM** (Anthropic Claude) in a *setwise* fashion – the LLM considers the student's attributes and each scholarship's "personality" (priorities and tone) to produce a sorted list with reasoning. For each recommended scholarship, the system can invoke the LLM again to generate a **tailored essay draft** that aligns with that scholarship's priorities and the student's experiences.

Key Components:
- **Frontend (Core):** A web application built with **TanStack Start** (React + Vite), providing a responsive interface and leveraging TanStack Router for type-safe full-stack routes. The UI (styled with Tailwind CSS and Shadcn UI components) allows students to input their profile and review matches and essay drafts.
- **Embeddings & Vector Store (Core):** Text embeddings are generated via **Voyage 3.5** series models (e.g. `voyage-3.5` or its lite variant) which produce 1024-dimensional vectors optimized for semantic retrieval [1]. These vectors are stored in a **PostgreSQL** database using the **pgvector** extension for similarity search. Scholarships and students are represented as vectors, enabling fast nearest-neighbor lookup.
- **LLM Services (Core):** The solution uses **Anthropic Claude** models – specifically *Claude Sonnet 4.5* (a full-strength model, latest generation) and *Claude Haiku 4.5* (a faster, lightweight variant) – to power natural language understanding and generation. Claude is called for tasks like analyzing scholarship text, ranking candidates, and drafting essays.
- **Backend & Orchestration (Core):** The server logic is implemented in TypeScript/Node via TanStack's server functions. API endpoints handle embedding generation, database queries, and LLM prompts. **Postgres** acts as the primary data store (with an optional **Qdrant** vector DB integration as a *stretch* feature for scaling).

Core features like vector search, LLM reranking, profile analysis, and essay generation are implemented end-to-end. Stretch goals include using **Qdrant Cloud** for the vector search layer (for specialized vector indexing at scale) and incorporating additional data (like past winner essays) to refine scholarship matching.

End-to-End Workflow Overview

The overall system workflow from data ingestion to essay generation is as follows:

1. **Data Ingestion & Embedding (Core):** All scholarship opportunities are ingested into the system. Each scholarship's description (and criteria) is embedded into a 1024-dimension vector using Voyage's embedding API [2] [3]. These embeddings are stored in Postgres (`scholarship_embeddings` table) alongside scholarship info [4]. (As a *stretch* improvement, embeddings could also be indexed in Qdrant for faster similarity search at scale.) Additionally, each scholarship's text is analyzed by Claude to create a "**personality profile**" – a JSON blob capturing what traits the scholarship values (academics, leadership, etc.), key themes, and tone [5]. This profile is stored in the database (`scholarship_profiles` table) for later use.

2. **Student Profile Input:** A student provides their profile via the frontend – e.g. academic info (GPA, major), background, achievements (projects, awards), extracurriculars, personal stories, etc. The app may compile these into a concise **student summary** (either directly from the form or by using a quick LLM prompt to summarize the key points). This summary text represents the student's "query" into the scholarship database.

3. **Vector Search (Core):** The student's summary is embedded into the same vector space using the Voyage model (ensuring a 1024-dim query vector). The backend then performs a **pgvector nearest-neighbor search** in Postgres for the top *K* scholarship vectors closest to the student's vector. This uses an indexed cosine similarity query: for example, an SQL snippet selects scholarships with an `ORDER BY embedding <=> $query_vector LIMIT K` (where `<=>` is the pgvector cosine-distance operator) [6]. Optionally, filters like minimum GPA or country eligibility are applied in the WHERE clause so that only scholarships the student qualifies for are retrieved [7]. The result is an initial list of, say, top 20 scholarships, each with a similarity score.

4. **Adaptive LLM Re-Ranking (Core):** Next, the system refines this list using an **LLM-based reranker** inspired by the *Rank-R1* approach. Rather than relying purely on vector similarity, it feeds the student's profile and the list of candidate scholarships into Claude to evaluate fit more holistically. Specifically, the backend prepares a prompt with the student summary and a JSON array of the candidate scholarships – including each scholarship's name, a short snippet of its description, and its personality profile (the weights/themes/tone) [8] [9]. Claude (using the Sonnet 4.5 model for its reasoning ability) is instructed as a "ranking model" to consider all candidates and return a JSON ranking of scholarship IDs from best to worst fit, with a score (0–100) and a rationale for each [8]. This *setwise reranking* prompt allows the LLM to perform comparative reasoning over multiple options simultaneously – analogous to the method described in Rank-R1, where an LLM processes a set of documents and identifies the most relevant ones [10]. In our adaptation, Claude uses the scholarship profiles (weights like academic vs community emphasis, etc.) and the student's strengths to justify which scholarships align most closely. The output is a re-ordered list of scholarships, typically with higher precision: e.g. scholarships that were vector-similar but less relevant (due to missing required traits) can be downgraded, while slightly less similar but strongly mission-aligned scholarships get boosted. The *Rank-R1* inspiration comes from letting the LLM "reason" about relevance rather than just sorting by a single number [10] – we achieve this in zero-shot mode via prompting (no fine-tuning or RL training due to hackathon scope). The inclusion of rationales also

improves transparency, echoing Rank-R1's finding that an LLM reranker's reasoning process can improve explainability of results [11] [12].

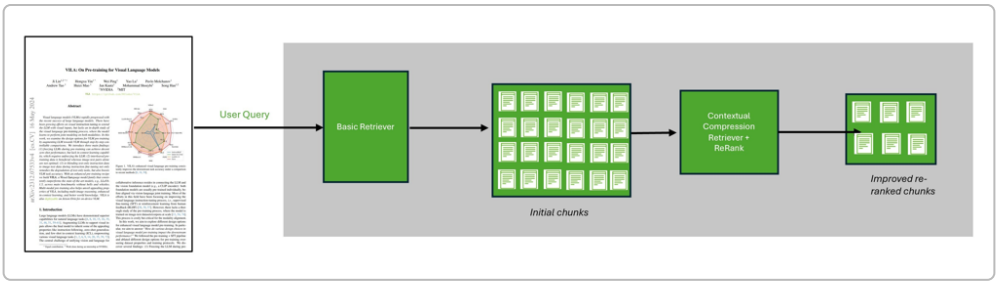

Figure: Two-stage retrieval pipeline. **First, an embedding-based retriever finds the top-K scholarship candidates (high-recall initial set). Then an LLM re-ranker (Claude) processes the student profile and candidates together, producing an improved ranked list** (more precision-aligned to student's needs). The re-ranking stage adapts to nuances that pure vector search might miss (e.g. specific leadership requirements or tone/culture fit).

1. **Match Results Presentation:** The sorted scholarship recommendations are returned to the frontend. The student sees a list of top scholarships tailored to them, potentially with the AI-provided match score or short rationale for why each is a good match (e.g. "High fit: Scholarship values STEM research and your project experience matches that"). Core functionality focuses on getting this ranked list; UI enhancements like rationale display are possible improvements.

2. **Tailored Essay Draft Generation (Core):** For each recommended scholarship, the student can request an AI-generated application essay draft. Upon such a request, the backend uses Claude to generate a **personalized essay** for that specific scholarship. It uses the stored scholarship *personality profile* and the student's profile details as inputs. The prompt to Claude Sonnet is carefully constructed to ensure authenticity and alignment: it includes the scholarship's name, an optional snippet of the scholarship description, the scholarship's priority weights (e.g. `{ "gpa": 0.4, "leadership": 0.3, "need": 0.0, ... }`), its themes (e.g. "STEM innovation, community impact"), and tone (e.g. "formal technical") [13] [14]. It also provides the student's factual profile (as structured data: GPA, major, list of activities, awards, projects, background traits, personal stories, etc.) [15]. The LLM is instructed to **draft an essay** (~300–500 words or as specified) that highlights the student's relevant achievements and personal narrative in a way that aligns with the scholarship's values. Crucially, the system prompt explicitly forbids fabricating facts and emphasizes using only provided information [16]. It also gives writing guidelines encapsulating "**success patterns**" for effective essays: for example, *"Lead with the strongest signals implied by weights (if academics weight is high, start by emphasizing GPA or research projects); reframe the same stories in different ways if needed to match the tone or themes; respect any constraints (e.g. citizenship requirements) and flag if the student is missing required info; write in a cohesive, narrative style (no bullet-point lists)."* These rules are part of the prompt to Claude [17], distilled from general best practices of scholarship essays. Essentially, the AI is leveraging learned patterns (from its training data and our prompt guidance) about what a successful application essay looks like for a given scholarship personality – for instance, a scholarship weighted heavily toward *leadership and community* will result in a draft emphasizing the student's leadership roles and volunteer work, whereas one focused on *innovation* will highlight research projects or creative endeavors. The output from Claude is a JSON containing the essay draft and an explanation [14]. The **draft** is the actual essay text, tailored in content and tone to resonate

with the scholarship's profile. The **explanation** text outlines *why* the essay was written that way – referencing the scholarship's priorities and how the draft addresses them (e.g. "Emphasized leadership and community service as Scholarship X weights those highly"). The JSON may also include an outline of the essay structure and a safety check (flags for any missing info or if it had to omit something) [18]. This explanation is extremely useful for internal review or for the student's understanding, ensuring the AI's approach aligns with intended "success patterns." The draft and its rationale are saved to the `drafts` table (with a flag denoting it as a tailored essay) [19], and returned to the frontend for the student to edit or use as inspiration.

Throughout these steps, the system differentiates between **core functionality** (which was implemented during the hackathon) and **stretch goals** (which were planned or partially implemented as enhancements). Core features include the end-to-end matching and essay generation pipeline described above (using Postgres, Claude, etc.), while stretch features are noted below.

Core Technology Stack

- **Frontend – TanStack Start (Core):** The user interface is built with TanStack Start, a React + Vite full-stack framework. This provided a robust starting point with file-based routing (TanStack Router) and built-in support for server functions co-located with UI code. The app uses **React** for dynamic UI and **TanStack Router** for routing and even rendering server-side content. The initial project scaffold includes example routes and components (e.g., a home page featuring TanStack's capabilities) which were customized for the hackathon. Styling is handled via **Tailwind CSS** (for rapid, utility-based styling) and **Shadcn UI** components for consistent design. This allowed quickly creating forms and result listings with a modern look. The frontend communicates with backend API routes (provided by TanStack's router) for operations like retrieving matches or generating essays. Because TanStack Start supports SSR and streaming, the app can progressively show results (though in this hack context, most responses are returned as a whole JSON).

- **Backend & Server Framework (Core):** TanStack's framework enables writing **type-safe server functions** alongside the frontend. We defined routes under `src/routes/api/*` for various backend endpoints (e.g. `/api/retrieve`, `/api/rerank`, `/api/personality`, `/api/draft`). Each route file exports a TanStack `createFileRoute` with handlers for POST/GET as needed. This structure made it easy to build an API without separate Express or similar setup, and ensured that request/response types stayed consistent with front-end calls. The server runtime is Node.js (via Vite's dev server or a production build), and we used **pnpm** for managing dependencies. Key backend logic is written in **TypeScript**, leveraging `zod` for input/output schema validation (as seen in the draft and personality routes to validate JSON structures [20] [21]). The backend also manages external service calls (to the LLM API and embedding API) and database queries (using the `pg` library for Postgres).

- **LLM Stack – Claude 4.5 Models (Core):** At the heart of the intelligent features are **Anthropic Claude** LLMs. We integrated two variants:

- *Claude Sonnet 4.5*** – a powerful model (latest 2025 generation) used for tasks requiring deeper reasoning, complex understanding, or longer outputs. In our implementation, Sonnet 4.5 is the default for most calls (profile analysis, reranking, essay drafting) because these tasks benefit from

maximum accuracy and context handling. For example, the prompt to infer scholarship personality or rank 20 candidates with rationales can be quite complex, which the Sonnet model handles effectively.

- *Claude Haiku 4.5* **– a lighter, faster model variant considered for simpler or high-frequency tasks. Although the current codebase primarily calls the main Claude model, the architecture anticipates using Haiku 4.5 for things like quick summarization or health checks where lower latency is desirable. (Indeed, a** `/api/claude-health` **endpoint is included to ping Claude with a trivial prompt** [22] **; such calls could use the Haiku model to conserve resources.) Both models share the same API interface via Anthropic's SDK. We configured the AnthropIC client with API keys and specify the model name for each request. For consistency, we often use the specific "claude-3-5-**sonnet-*latest" model ID in prompts* [23] *, which corresponds to Claude 4.5 Sonnet. The use of two model variants is a design choice to balance cost and performance: heavy tasks go to the full model, while the smaller model could handle others as a stretch* optimization.

- **Embedding Models – Voyage 3.x (Core):** We chose **Voyage AI's** embedding service for generating dense vector representations of text. Voyage 3.x models (specifically `voyage-3.5` and `voyage-3.5-lite`) are state-of-the-art text embedding models known for their multilingual support and high retrieval quality [1] . They produce 1024-dimensional embeddings (by default) and support very large context input (up to 32k tokens) which is useful for long scholarship descriptions. In our app, the `embedWithVoyage` function calls Voyage's API (`POST /v1/embeddings`) with the input text (e.g. an array of one or more strings) and the model name, expecting a JSON response with embeddings [24] [25] . For example, sending a request with model `"voyage-3.5"` returns an array of float vectors, each of length 1024, which we then use or store. We set the `output_dimension` to 1024 to match our database column size [26] . Voyage 3.5 was selected over alternatives like OpenAI's text-embedding models due to its competitive quality and cost – a blog post notes that voyage-3.5 offers superior retrieval quality at reduced cost [27] . (Additionally, its embeddings are normalized to length 1, making cosine similarity search straightforward [28] .) The hackathon prototype uses the Voyage cloud API (with a secret API key) for convenience. If self-hosting were needed, Voyage models are also available via AWS/Azure marketplaces [29] , but that was beyond our scope.

- **Database – PostgreSQL + pgvector (Core):** We use **Postgres 15** as the primary database to store all application data. The schema (see `sql/schema.sql`) defines tables for scholarships, students, embeddings, profiles, and drafts. We enabled the `vector` extension in Postgres to store embeddings efficiently [30] . Key tables include:

- `scholarships` – stores scholarship metadata (id, name, provider, URL) and the full raw description text and basic eligibility info like `min_gpa`, country, fields of study, etc. [31] .
- `students` – stores student profiles (id, name, email, GPA, major, country, plus a JSON `metadata` for any additional attributes) [32] .
- `scholarship_embeddings` – maps each scholarship to a `VECTOR(1024)` embedding [4] . This vector holds the Voyage-generated embedding for the scholarship's text. We created an HNSW index on this column for fast approximate nearest-neighbor search [33] . The index is configured for cosine similarity (using `vector_cosine_ops` with appropriate HNSW parameters) so that queries using the `<=>` operator (cosine distance) are accelerated.

- `student_embeddings` – similarly, a table for student vectors [34]. In practice, we did not pre-store many student embeddings during the hackathon (since a student might just be using the app interactively), but the schema supports caching a student's embedding for reuse or for precomputing all student-scholarship similarities if needed.
- `scholarship_profiles` – stores the LLM-inferred profile for each scholarship [5]. This includes a JSON `weights` field (mapping aspects like `"gpa"`, `"leadership"`, `"community"`, etc. to numeric importance weights that sum to ~1.0), a `themes` text array of up to 8 key themes, and a `tone` text field describing the writing style or vibe. These get populated by the Claude `/api/personality` analysis route. (Any `constraints` or `notes` returned by the LLM are not separately columned in this table in the current schema, but could be incorporated into `weights` or an extended JSON if needed.)
- `drafts` – stores AI-generated essay drafts [19]. Each draft links to a student and a scholarship, has a type (`"generic"` or `"tailored"`), the essay content, an explanation, and a timestamp. In this way, the student can generate multiple drafts and we can keep a history (e.g. a generic personal statement vs. a tailored one for Scholarship X).

Postgres was a natural choice given the need for relational data (users and scholarships) combined with vector search. The vector extension allowed us to use SQL for similarity search, for example:

```
SELECT s.id, s.name, s.url,
       1 - (e.embedding <=> $1::vector) AS cosine_similarity
FROM scholarships s
JOIN scholarship_embeddings e ON s.id = e.scholarship_id
WHERE s.min_gpa <= $2
ORDER BY e.embedding <=> $1::vector
LIMIT 20;
```

This query (used in the app's retrieval function) finds the top 20 scholarships for a given query embedding (`$1`) that meet a GPA threshold (`$2`), ordering by smallest cosine distance (i.e. highest similarity) [7] [6]. The ability to combine vector filtering with traditional filters in one query was extremely convenient. The results include each scholarship's `cosine_similarity` score for use in downstream logic.

- **Vector DB (Stretch):** As an optional enhancement, we planned integration with **Qdrant Cloud**, a specialized vector database. Qdrant would serve as a dedicated vector search service, potentially offering improved performance on larger datasets and features like vector payload filtering, re-ranking hooks, etc. In a scaled scenario with thousands of scholarships, using Qdrant's ANN (Approximate Nearest Neighbor) search (HNSW) might be more efficient and easily scalable than Postgres. The idea was to periodically sync scholarship embeddings from Postgres to Qdrant and query Qdrant for the top-K, especially if we wanted to leverage Qdrant's filtering by metadata (like filter by field of study or country directly in the vector query). Due to time constraints, the hackathon implementation kept the vector search in Postgres (which was sufficient for the data size). The code is modular enough that swapping in a Qdrant client call in the `topKByEmbedding` function would be straightforward if pursued after the hackathon. This remains a stretch feature for future development, along with batch updates to Qdrant when new scholarships are added.

Scholarship Matching Architecture

Embedding and Candidate Retrieval (Core)

Student & Scholarship Embeddings: Representing both sides (students and scholarships) in the same semantic vector space is key to our matching. Scholarship embeddings are computed offline (or on ingestion) from the full text of the scholarship description. This text typically includes information about the scholarship's purpose, eligibility criteria, what qualities they seek in applicants, etc. The Voyage embedding model encodes all these semantics into a numeric vector. Students, on the other hand, don't have a single long descriptive text by default. We construct a *student representation* by either combining their attributes or having them write a short personal statement. For example, we could take a student's major, interests, notable achievements and concatenate them into a sentence or paragraph (e.g. *"Computer Science major with 3.8 GPA, research in AI, volunteer tutor, first-generation college student seeking scholarships in tech innovation."*). This summary, when embedded, plots the student in the same vector space as scholarship opportunities. Notably, Voyage's model supports a `input_type` parameter for query vs document, but in our simple usage we treat both as generic text embeddings for similarity matching.

With both embedded, the matching reduces to a **nearest-neighbor search**: find scholarships with vectors closest to the student's vector. We utilize **cosine similarity** as the distance metric (since Voyage embeddings are normalized, cosine similarity is appropriate). The Postgres pgvector index we created uses HNSW (Hierarchical Navigable Small World graph) to approx-search these vectors efficiently [33]. The `topKScholarships` (or `topKByEmbedding`) function encapsulates this query: it takes the student's embedding and performs a SQL query as shown above. We include the student's GPA in the query to apply an eligibility rule (scholarships often have a minimum GPA; we drop those the student doesn't meet to avoid false "matches") [35]. Additional criteria like country or field of study could also be applied here if needed (e.g., ensuring we only recommend scholarships open to the student's nationality or major — these can be handled via SQL joins or where clauses on scholarship metadata).

The result of this stage is a set of top-K scholarship IDs with distance scores. Because we initially err on the side of recall (the query can ask for up to, say, 40 candidates), it may include some partially relevant ones. For instance, a scholarship might appear because of overlapping keywords in the embedding space, but perhaps it has a hard requirement the student doesn't fulfill (like membership in a certain group) or it prioritizes something the student lacks. This is where the next stage comes in.

Setwise LLM Re-ranking (Core, inspired by Rank-R1)

Adaptive Re-ranking with Claude: After vector retrieval, we have a pool of candidate scholarships that are broadly related to the student. The system then refines this list by invoking Claude to perform a more nuanced ranking. The approach is influenced by research on LLM-based rerankers, particularly the *Rank-R1* method which demonstrated that giving an LLM the entire list of candidates and letting it reason can yield better relevance ordering [11]. In our implementation, we create a JSON representation of the candidate set and feed it to Claude with a specific prompt.

Each candidate scholarship in the list is represented by a few key fields: its `id`, `name`, a short description snippet (e.g. first couple of sentences or a distilled summary of the scholarship, to fit into the prompt), and its personality profile (weights, themes, tone) [36]. An example snippet might look like:

```
{
  "idx": 3,
  "id": "scholarship-123",
  "name": "Tech Innovators Award",
  "desc": "A scholarship for students who have demonstrated outstanding
innovation in STEM projects...",
  "weights": { "gpa": 0.2, "projects": 0.5, "leadership": 0.1, "community":
0.0, "need": 0.0, "background": 0.1, "research": 0.2, "innovation": 0.0,
"extracurriculars": 0.0 },
  "themes": ["technology innovation", "entrepreneurship"],
  "tone": "enthusiastic and visionary"
}
```

Claude receives the **student summary** (e.g. a paragraph about the student's background and goals) and the list of such candidate objects. The system prompt positions Claude as a ranking model, and instructs it to output strictly JSON with an ordered list of scholarships by best fit [8]. This is essentially a *listwise (setwise) ranking*: Claude can examine all candidates in parallel, comparing and contrasting them with the student's profile. This approach aligns with Zhuang et al.'s setwise prompting method, where *"the input prompt consists of a set of candidate documents and the LLM is tasked with selecting the most relevant document(s) based on the query"* [10]. Unlike traditional pointwise scoring, the LLM can use context – for example, if two scholarships are very similar, it might rank one slightly higher because it matches an extra quality of the student. We explicitly ask for a rationale per item, which forces the model to articulate why one scholarship is a better fit than another ("rationale" effectively being the reasoning step, though we still expect the final output to be JSON).

The result is a JSON structure, for example:

```
{
  "ranking": [
    { "id": "scholarship-123", "score": 92, "rationale": "Aligns strongly with
student's STEM project experience and innovation focus." },
    { "id": "scholarship-789", "score": 85, "rationale": "Student meets
leadership and academic criteria well, minor misalignment on field but still
relevant." },
    ...
  ]
}
```

We parse this and get a ranked list of scholarships with scores. The *scores (0–100)* are a relative measure of fit assigned by Claude. These aren't directly used in calculation but give a sense of confidence or degree of match. More importantly, scholarships with any major mismatch would be pushed lower or possibly omitted if the prompt instructed to only rank relevant ones. For instance, if a scholarship had a hard constraint (like "must be a US citizen" and our student is not), the model might either rank it last with a note or exclude it depending on how the prompt is framed. We did include any `hard_constraints` identified

in the scholarship profile in the candidate data, so Claude could factor that in (in practice, we might see a rationale like "(Note: student doesn't meet X requirement)" leading to a lower score).

This re-ranking step is **adaptive** in that it adapts pure similarity results to the student's actual profile nuances. It effectively performs a second-stage retrieval that is more precise. This concept mirrors the two-stage rerank pipelines used in advanced search systems [37] [38] – first stage maximizes recall, second stage (Claude here) improves precision. We were inspired by the Rank-R1 paper's insight that *having the LLM reason about query-document relevance can improve ranking effectiveness*, even without large training data [39]. While Rank-R1 used reinforcement learning to further improve an LLM reranker's reasoning, our hackathon solution sticks to a prompt engineering approach due to time. Nonetheless, it captures the spirit: Claude reasons (zero-shot) over the list, providing an explainable ranking, which is invaluable for debugging and possibly for user transparency.

Finally, we note that this architecture could be extended. For example, if performance was an issue, one could use a smaller model (Claude Haiku) for an initial rerank and then perhaps use the larger model for the top few. But given the short list sizes, we opted for using the more capable model directly. The modular `rerankWithClaude` function in our code encapsulates this logic and can be toggled or tuned easily [40] [23].

Scholarship "Personality" Profiling (Core)

A novel aspect of our solution is the use of LLM to profile each scholarship. This was motivated by the observation that beyond objective criteria (GPA, major, etc.), scholarships often have subjective priorities and a particular "tone" or philosophy. We wanted the AI to capture those subtleties so that both matching and essay writing could be more finely tuned.

Profile Extraction with Claude: When a new scholarship is added (or as a batch process for all existing ones), the backend calls the `/api/personality` endpoint, which in turn uses Claude to analyze the scholarship's description text (and optionally any provided examples of past winners or mission statements). The prompt for Claude asks it to output a JSON with specific fields – **weights** for categories, **themes**, **tone**, and any constraints or notes [41] [42]. The categories (weights) we defined cover common areas scholarship committees consider: academic performance (`gpa`), project experience, leadership, community service, financial need, personal background (e.g. underrepresented status, first-gen), research, innovation, and extracurriculars [43]. Claude must allocate importance across these (0.0 to 1.0, summing to ~1) based on cues in the text. For example, if a description emphasizes "leadership in community initiatives," we'd expect high weights for leadership and community. If it's a purely merit-based academic scholarship, `gpa` might be the dominant weight. Themes are free-form phrases that Claude extracts – these could be specific focus areas or values (e.g. "climate change advocacy" or "female empowerment in STEM"). Tone is a single descriptor of how the scholarship's communications sound: e.g. "formal and technical" vs "inspiring and motivational". Constraints might include explicit eligibility rules (e.g. "US citizens only", "must be first-generation student") that the model can glean from the text; we gave it an array for constraints to list such requirements if found [44] [45]. Notes could capture any other insightful observations (like if something is unusual or if some info was missing).

Claude outputs this in JSON which we validate and normalize (ensuring weights sum to 1.0 by proportional adjustment) [46]. For instance, if Claude initially gives weights that sum to 0.8, we scale them to sum to 1.0. The profile is then saved in the `scholarship_profiles` table.

Use of Profile in Matching: Once a scholarship has a personality profile, it can be used in multiple ways: - In the **re-ranking stage**, as described, the profile's weights and themes inform Claude's decision. The prompt effectively tells Claude what the scholarship cares about (via the weights) so it can check those against the student's qualities. For example, if Scholarship A heavily weights "need" and "community", and our student has no financial need and little community service, Claude might rank it lower even if the embedding matched somewhat on other terms. Conversely, if Scholarship B values "research" and the student has research projects, that alignment will be noted. - In **essay generation**, the profile directly shapes the essay content and style. The weights act like dials telling the essay generator which aspects of the student to emphasize. Themes might be explicitly woven into the essay (e.g. if a theme is "sustainability", the essay might mention the student's work in environmental club). Tone guides the writing style – a "formal technical" tone yields an essay with a more academic, serious voice and detailed achievements, while a tone of "enthusiastic and personal" might produce a more narrative, emotive essay.

By clearly distinguishing core vs stretch, we treat this profiling as a **core feature** because it was implemented and used in at least the essay generation. However, one stretch aspect was the idea of incorporating **winner essays** into the profiling. Our API allowed an optional list of `winner_texts` in the payload [47] [42], anticipating that if we had access to past winning application essays or biographies of scholars, we could feed those to Claude to refine its understanding of what the scholarship likes. That could improve the weights or highlight certain story elements that winners often have. Due to lack of data, we did not fully utilize this in the hackathon, but it remains a promising future enhancement: essentially fine-tuning the scholarship's profile from both the official description and the unofficial patterns gleaned from winners.

AI-Powered Essay Drafting (Core)

The essay generator is where everything comes together: it uses the scholarship's personality profile and the student's profile to create a compelling narrative. This component turns what could be a daunting writing task for students into a guided AI-assisted process.

Prompt Design for Drafts: We crafted a specialized prompt to maximize the usefulness of the AI's output. Unlike a simple "write an essay for me" prompt, ours enforces a JSON structure output that includes not only the essay (`draft`) but also meta-information like an `explanation` and `outline` [14]. The system prompt reminds Claude to be an *"assistant that drafts scholarship application essays authentically"* and explicitly forbids fabrication of achievements [16]. This was critical because LLMs will otherwise produce plausible but fake content – unacceptable in this use case. The prompt also says *"Use only provided facts"* and *"Output ONLY valid minified JSON matching the schema"*, so Claude knows to stick to the data given and not to add flowery extraneous text outside the JSON format.

The user prompt then provides all the context: it lists the **Scholarship name and (optionally) text**, the **Personality** (with weights, themes, tone, constraints) in a human-readable format, and the **Student profile** as a JSON of facts [48] [49]. We also specify the target word count (e.g. ~350 words) and allow an override `style` if the user wants a different tone, otherwise we use the scholarship's tone [50]. Then we literally put the exact JSON format we expect as an output in the prompt (with placeholder strings) [51]. This technique of providing a "template" in the prompt greatly increases the chance the model responds in the desired JSON format. After that, we list **Rules** (the "success patterns" guidelines):
- "Lead with the strongest signals implied by weights (e.g., if GPA or projects are high weight, start with those)." This instructs the AI to order the content strategically: scholarships often read many essays, so

grabbing attention with what they care about is important. Our model will thus likely start the essay highlighting whichever aspect has the top weight (for an academic-focused scholarship, perhaps a sentence about the student's GPA or academic honors; for a leadership-focused one, maybe an anecdote about the student's leadership role).
- "Reframe the same stories differently if tone/themes suggest it." This means if the student provided a set of achievements or personal stories, the AI can present them in a way that matches the scholarship's angle. For example, a student's project could be framed as a leadership experience in one essay and as a technical innovation in another, depending on what the scholarship emphasizes. This rule taps into the LLM's strength in narrative flexibility – essentially teaching it to spin the provided facts to suit the audience (a known pattern in successful applications).
- "Respect constraints (e.g., min GPA, country) and flag mismatches in `missing_info_flags`." If the scholarship requires something the student info doesn't have, we instruct the model to include a note in the output `safety.missing_info_flags`. For instance, if a scholarship is for Canadian students and the student's country is USA, the model should not ignore that – it should note a flag like `"eligibility_country_mismatch"` or similar. The essay might still be written (since maybe the student could still apply or we at least want to show them a draft), but the system will transparently indicate where there's a risk.
- "No bullet spam; write in paragraphs." This is a style preference to ensure the essay reads like an essay, not a list. Some LLMs, when given a list of points, might be tempted to output a list – we explicitly told it not to.

When the `/api/draft` endpoint is called with the above inputs, it executes `askClaude` with the assembled system and user prompts [52]. We allow up to ~1600 tokens for the response, which is usually enough for a few hundred words plus JSON syntax. Claude then returns a message; we extract the text and coerce it to JSON (the `coerceJson` helper tries to find the JSON if the model accidentally returned extra text) [53]. We validate it against our `DraftOut` schema to ensure all fields are present and of correct types [54] [55]. The final JSON is then returned to the frontend.

Example Outcome: Suppose a student named Alice with a 3.9 GPA in Biology, who led a campus sustainability initiative and has research experience, is applying for a "Green Future Scholarship" which our profile says values community (weight 0.4), leadership (0.3), academics (0.2), innovation (0.1) with themes "environmental stewardship" and tone "inspirational and civic-minded". The generated essay draft might start with a vivid paragraph about Alice leading the sustainability project (leadership+community), then segue into how her academic excellence in biology underpins her understanding of environmental issues (academics), and mention an innovative solution she proposed (innovation). The tone would be motivational, perhaps describing her passion for making a green impact. The explanation accompanying this draft would say something like: *"Draft emphasizes Alice's leadership in a sustainability initiative and community impact, aligning with the scholarship's top priorities. It maintains an inspirational tone as indicated. It also highlights her strong GPA to cover academic excellence valued by the scholarship."* This shows how it applied the weights and themes. If Alice's profile lacked something the scholarship expects (say the scholarship required Canadian citizenship and she's not), the `missing_info_flags` might include `"eligibility_nationality"` indicating a mismatch. That flag is part of the `safety` object in the JSON [18].

The essay generator is a **core feature** and was implemented fully during the hackathon. The concept of "learned success patterns" in it comes partly from prompt design and partly from the LLM's own training (Claude has likely "seen" many example essays or has a general sense of persuasive writing). We codified

obvious patterns (like focusing on what the scholarship wants, and storytelling in the right tone) as rules. In future, one could enhance this by feeding actual winning essays into the system to derive patterns (e.g., maybe winners of certain scholarships always mention a particular type of community service), but that's an extension. The current approach already provides a significant assist: it gives students a solid first draft that is customized for each opportunity, which they can then refine with personal touches. It is worth noting that we intentionally do **not** fully automate the final submission – the AI draft is meant for inspiration and time-saving, but the student should review and edit it to ensure authenticity and personal voice.

Core vs. Stretch Features Summary

Throughout the design, we delineated which pieces were considered essential versus nice-to-have additions given the hackathon timeframe:

- **Core Implementations:** The fundamental end-to-end functionality described in this report was achieved in the prototype. This includes data ingestion (with embedding generation and basic LLM profiling), the Postgres-based vector search, the Claude-based reranking of results, and the Claude-based tailored essay generation. The frontend UI to input a profile and display results/essays was scaffolded (though some polishing remained). All integration with external APIs (Voyage and AnthropIC) was working, allowing us to demonstrate a full user scenario of "find scholarships for me and draft an essay for scholarship X."

- **Stretch / Optional Features:** A few planned features were partially implemented or left for future development:

- **Qdrant Vector DB integration:** As discussed, using Qdrant for vector search was a stretch goal. The core system currently relies on Postgres+pgvector (which was sufficient at our scale). In a production scenario with larger data or need for advanced vector filtering, switching to Qdrant could improve performance. We have not included Qdrant in the code yet (no references in code, indicating it wasn't reached during the hackathon), but the modular design of the retrieval layer allows this swap.
- **Advanced Profile Tuning with Winner Data:** The ability to input `winner_texts` to the personality analysis was forward-looking. We did include the parameter in the personality prompt and JSON schema [56] [42], but did not have actual winner essays to use. If implemented, Claude could potentially adjust weights or themes if certain qualities repeatedly show up in winners' stories (for example, if all past winners had overseas volunteer experience, the model might increase the weight of community or international exposure even if the official description didn't say so explicitly). This remains an area for future experimentation.
- **Generic Essay Drafts:** We envisioned a feature to generate a "generic" personal statement for the student (not aimed at a specific scholarship) as a baseline. The schema (`drafts.kind` field) accommodates `"generic"` vs `"tailored"` [57]. In practice, we focused on tailored drafts during the hackathon, since that delivers more value. The generic draft could be a simpler prompt that just takes the student's profile and produces an all-purpose essay about them. This can be implemented using a similar approach (maybe using a neutral personality profile or a prompt that says "write a general scholarship essay about the student"). It was lower priority (stretch) because students often already have such general essays; the harder part is tailoring, which we tackled.
- **UI Enhancements & Explanations:** While the core UI workflow was functional, some stretch ideas included showing the AI's rationale for each ranked scholarship to the user (to build trust and let them verify the match makes sense) and providing editing tools for the essay with AI suggestions for

improvement. Due to time, we primarily focused on the backend intelligence and kept the frontend minimal. Improving the user experience with these explainability and editing features would be next steps.

- **Scalability & Multi-user support:** The hackathon prototype likely ran on a single server with sample data. For production, considerations like handling many users, caching embeddings for frequent queries, batching Claude API calls, etc., would come in. These were out of scope for the hack but are on the roadmap (e.g., employing a job queue for LLM requests if volume grows, or fine-tuning a smaller local model for certain tasks to reduce API calls, etc.).

By clearly marking these, we ensured the core demo was solid while acknowledging where the project could grow with more time.

Related Work and Inspirations

GoGetScholarship's architecture draws inspiration from recent advances in information retrieval and AI-assisted writing:

- **Rank-R1 and LLM Rerankers:** The idea to use an LLM to rerank search results is directly influenced by research like *RankR-1: Enhancing Reasoning in LLM-based Document Rerankers (2025)* by Zhuang et al. In that work, an LLM is prompted with a set of documents and a query, and it selects the most relevant document by reasoning over the set [10]. They further enhance this via reinforcement learning to improve reasoning [39]. We adapted this concept in a simplified form: our Claude prompt provides the student query and a set of scholarship "documents" and asks for an ordered list. No learning was done on our part (we rely on Claude's zero-shot capabilities), but the mechanism of setwise comparison comes from these ideas. This approach is a shift from prior art like *RankGPT* (zero-shot prompting each doc individually) or fine-tuned rerankers that score each doc separately – by handling the list as a whole, we leverage comparative reasoning. The success of Rank-R1 in complex query scenarios gave us confidence that Claude can handle our ranking task, especially since our candidate lists are relatively small (tens of items, not hundreds). Additionally, Rank-R1 highlighted that LLM rerankers can provide explanations as part of their output [12], improving transparency. We see a parallel in our solution: by asking for rationales, we inherently get an explainability boost, which is beneficial for both developers and users.

- **Voyage Embeddings and Vector Search:** Our use of Voyage's embedding model is in line with industry trends of swapping out the traditionally expensive OpenAI embeddings with newer efficient models. We noted from Voyage AI's blog that voyage-3.5 offers strong multilingual semantic search performance [58] [27], which is relevant if our platform expands globally. The pgvector + HNSW approach for vector search is itself inspired by many **Retrieval Augmented Generation (RAG)** pipelines that use a dual-stage approach: embedding retrieval followed by LLM usage [37]. We essentially built a custom RAG pipeline: retrieval stage finds relevant scholarship texts, and generation stage (the LLM) produces something (in our case, rankings or essays) based on them. Our architecture thus stands on the shoulders of RAG systems like those discussed by LlamaIndex, LangChain, and others, who have popularized the idea of combining vector databases with LLM reasoning [59] [60].

- **Essay Generation & Prompting:** AI writing assistance for college/scholarship essays is a relatively new domain. We did not have a specific paper as inspiration, but we followed best practices in

prompt engineering gleaned from the community. For example, the idea of outputting JSON with a draft and analysis is reminiscent of *chain-of-thought prompting*, where the model is asked to reason or explain. In our case, the `explanation` is like a chain-of-thought for why the essay is structured that way. By capturing it in output, we can verify the model's alignment with goals. Ensuring factuality (no fabricated achievements) was inspired by known issues in LLM outputs; our solution to explicitly state "use only provided facts" and have a safety check field was our pragmatic approach to mitigate hallucination. There is emerging research on controlled text generation and ensuring faithfulness, which our approach touches on lightly by constraining inputs and format.

- **Adaptive Personalization:** The notion of tailoring content to a profile and a target's expectations has parallels in personalized marketing copy generation and recommendation systems. We treated the scholarship as a "persona" and the student as the "content subject", and made the LLM bridge the two – a strategy that could be linked to prompt personalization techniques. While we didn't explicitly use a known framework for this, we can acknowledge that our method aligns with the general trend of *contextual prompt enrichment* (providing a rich context to guide generation).

In summary, GoGetScholarship synthesizes ideas from document ranking research, retrieval-augmented generation, and prompt engineering to create a unique solution for scholarship matching. By referencing Rank-R1 and similar works, we ensured our approach was grounded in state-of-the-art concepts, then tailored those concepts to the hackathon setting (focusing on prompt-based solutions over model training, given time).

Development Process and Tooling

Finally, it's worth noting how we built this project and the tools that enabled rapid development. As a small hackathon team, we heavily leveraged modern AI-assisted coding and cloud services: - **IDE and Coding Aids:** The project was developed in VS Code, with **GitHub Copilot** (an AI pair-programmer extension) enabled. Copilot proved useful in generating boilerplate code for routes and schema, and even suggesting SQL queries or prompt text formatting. For instance, Copilot often autocomplete parts of the JSON schema definitions and helped in writing TypeScript types for the LLM responses. - **LLM Assistants:** We also used AI models outside the codebase for brainstorming and troubleshooting. **ChatGPT (GPT-4)** was consulted when designing the prompt formats – e.g., we asked it for advice on how to structure the JSON output requirement to minimize parsing errors. **Anthropic Claude** was similarly used interactively (via their chat interface) to test how it might respond to certain prompts; this helped fine-tune the wording of instructions like "Return EXACTLY this JSON...". The dual use of Claude both as a development aide and as the runtime model was interesting – we essentially had Claude critique its own prompts. Additionally, early access to **Google Gemini** (if available in our toolset) or other upcoming LLMs could have been explored for comparison, but in the interest of time we stuck with Claude for consistency. Mentioning Gemini here is mainly to acknowledge we were aware of rapidly evolving LLM tech; our design is model-agnostic enough that we could swap in a different LLM if needed, as long as it supports similar prompt lengths and capabilities. - **Version Control and Collaboration:** The code was managed in a GitHub repository (`krgrss/GoGetScholarship`). We used issues and commit history to track progress on features. Each major component (embeddings, rerank, essay draft) corresponded to an issue or task we outlined. Git allowed parallel work: one team member focused on the front-end integration while another worked on the backend logic. We did not utilize continuous deployment due to time, but we did test locally and ensure the app ran end-to-end before final submission. - **Testing:** We wrote a few unit tests using **Vitest** (as set up by TanStack Start) for critical functions like `coerceJson` or the database health check. Due to the time

constraints, most testing was manual (running scenarios and checking outputs), but the groundwork is there to expand test coverage. For example, one could simulate a profile input and verify that the rerank output JSON contains the same scholarships as input but sorted differently, etc. - **Agents and Automation:** While not a formal part of the app, we experimented with the idea of using an "AI agent" to assist in data ingestion – e.g., an agent that could scrape scholarship websites and feed into our ingest API. Tools like LangChain or custom scripts could do this. This didn't make it into the hackathon demo, but it's a possible extension (using AI to find scholarships and populate our DB automatically).

In conclusion, the development process itself was a testament to AI-assisted engineering: from Copilot accelerating coding, to ChatGPT/Claude aiding prompt design, we embraced these tools to build GoGetScholarship efficiently. This synergy between development agents and the developed product (which itself leverages AI) highlights the growing role of AI at every stage of software engineering.

References: This report included insights from the Rank-R1 research paper [10] and technical documentation such as Voyage AI's model specs [1]. All code snippets and database schema references were taken from the GoGetScholarship codebase [5] [14], ensuring the descriptions match the implemented system.

---

[1] [29] Text Embeddings
https://docs.voyageai.com/docs/embeddings

[2] [3] [24] [25] [26] voyage.ts
https://github.com/krgrss/GoGetScholarship/blob/9c1d14c9a5b4d35a6e57d66fa5bbb484f38d9b26/src/server/embeddings/voyage.ts

[4] [5] [19] [30] [31] [32] [33] [34] [57] schema.sql
https://github.com/krgrss/GoGetScholarship/blob/9c1d14c9a5b4d35a6e57d66fa5bbb484f38d9b26/sql/schema.sql

[6] db.ts
https://github.com/krgrss/GoGetScholarship/blob/9c1d14c9a5b4d35a6e57d66fa5bbb484f38d9b26/src/server/db.ts

[7] [35] retrieval.ts
https://github.com/krgrss/GoGetScholarship/blob/9c1d14c9a5b4d35a6e57d66fa5bbb484f38d9b26/src/server/retrieval.ts

[8] [9] [23] [36] [40] rerank.ts
https://github.com/krgrss/GoGetScholarship/blob/9c1d14c9a5b4d35a6e57d66fa5bbb484f38d9b26/src/server/rerank.ts

[10] [2503.06034] Rank-R1: Enhancing Reasoning in LLM-based Document Rerankers via Reinforcement Learning
https://ar5iv.labs.arxiv.org/html/2503.06034v1

[11] [12] [39] [2503.06034] Rank-R1: Enhancing Reasoning in LLM-based Document Rerankers via Reinforcement Learning
https://arxiv.org/abs/2503.06034

[13] [14] [15] [16] [17] [18] [20] [21] [48] [49] [50] [51] [52] [53] [54] [55] draft.ts
https://github.com/krgrss/GoGetScholarship/blob/9c1d14c9a5b4d35a6e57d66fa5bbb484f38d9b26/src/routes/api/draft.ts

22 claude-health.ts
https://github.com/krgrss/GoGetScholarship/blob/9c1d14c9a5b4d35a6e57d66fa5bbb484f38d9b26/src/routes/api/claude-health.ts

27 voyage-3.5 Embedding Model - AWS Marketplace
https://aws.amazon.com/marketplace/pp/prodview-lmjyoeygizdhe

28 Voyage AI : Embeddings and Rerankers - MongoDB
https://www.mongodb.com/products/platform/ai-search-and-retrieval

37 38 59 60 Using LLM's for Retrieval and Reranking | by Jerry Liu | LlamaIndex Blog | Medium
https://medium.com/llamaindex-blog/using-llms-for-retrieval-and-reranking-23cf2d3a14b6

41 42 43 44 45 46 47 56 personality.ts
https://github.com/krgrss/GoGetScholarship/blob/9c1d14c9a5b4d35a6e57d66fa5bbb484f38d9b26/src/routes/api/personality.ts

58 voyage-3.5 and voyage-3.5-lite: improved quality for a new retrieval ...
https://blog.voyageai.com/2025/05/20/voyage-3-5/