[Indy-1] - Next Mission Navigator

AI 7993 – Section W01 – Spring 2026

February 1, 2026

Crystal Tubbs

Team Members:

Name	Role	Cell Phone/Alt Email
Crystal Tubbs	AI Engineer, Full-Stack Developer, Prompt Engineer, Voice AI Integrator	(727)686-6544 Onmybutterflyjourney@gmail.com
Arthur Choi	Project Advisor	(770) 867-5309 Achoi13@kennesaw.edu

Software Design Document

Table of Contents

1 Introduction and Overview 2
1.1 Document Outline 2
1.2 System Introduction 2
2 Design Considerations 2
2.1 Assumptions and Dependencies 2
2.2 General Constraints 3
2.3 Development Methods 3
3 Architectural Strategies 3
4 System Architecture 4
4.1 Subsystem Architecture 4
5. Detailed System Design 6
5.1 Classification 6
5.2 Component Definitions 7
5.3 Component Responsibilities 8
5.4 Constraints 9
5.5 Composition 10
5.6 Uses/Interactions 11
5.7 Resources 11
5.8 Processing 12
5.9 Interface/Exports 14
5.10 Detailed Subsystem Design 15
5.10.1 Frontend 15
5.10.2 Backend API 16
5.10.3 Intelligence Engine 18
5.10.4 Knowledge Curation Layer 19
5.10.5 Voice Module (Optional) 20
6 Operational Scenarios 21
6.1 Standard Plan Generation Flow 21
6.2 Error Handling Scenarios 23
7. Glossary 25
8. Bibliography 26

1 Introduction and Overview

1.1 Document Outline

This Software Design Document (SDD) provides a comprehensive description of how NextMission Navigator will be constructed. It translates the functional and non-functional requirements defined in the Software Requirements Specification (SRS v1.0) into a set of design artifacts that can guide implementation, testing, and future maintenance. The document is organized according to recommended practice for software design descriptions, starting with an overview and progressing through design considerations, architectural strategies, system architecture, detailed design, operational scenarios, glossary, and bibliography.

This SDD is intended for system architects, developers, testers, and stakeholders who need to understand how the system is structured. It does NOT include deployment procedures, database administration guides, or end-user training materials; those are covered in separate operations and training documentation.

1.2 System Introduction

NextMission Navigator is a web-based decision-support platform that assists U.S. military veterans in planning their transition to civilian life. Users enter a personal goal (such as "find a job in cybersecurity" or "pursue a degree in mechanical engineering") along with optional context (branch of service, years served, and location). The system validates the input, augments the prompt with curated regional resources, invokes a large language model (LLM), and returns a structured action plan comprising prioritized steps, descriptive titles, category classifications, and metadata.

The system consists of five primary subsystems: a React/Next.js frontend for user interaction, a Backend API layer handling orchestration and validation, an Intelligence Engine that constructs prompts and invokes the LLM, a Knowledge Curation layer containing verified regional resource datasets, and an optional Voice Module for text-to-speech audio playback. The system is deployed on Vercel's serverless platform and communicates with external services (Anthropic Claude API and ElevenLabs TTS) over HTTPS.

2 Design Considerations

2.1 Assumptions and Dependencies

The design of NextMission Navigator relies on the following assumptions and dependencies:

Internet connectivity: Users must have access to the internet and a modern web browser (Chrome, Safari, Edge, or Firefox). The backend assumes reliable network connectivity to the LLM and voice APIs.

User input: The system assumes that users will provide a clear goal and optional context (branch, years of service, location). Input validation and sanitization guard against injection attacks and malformed data.

Third-party services: The platform depends on Anthropic Claude API for plan generation and ElevenLabs API for voice synthesis. Service interruptions are mitigated through retry logic (exponential backoff, max 3 attempts), response caching (15-minute TTL for repeated queries), and graceful degradation (core plan generation continues if voice service fails).

Regional datasets: Accurate regional resource datasets must be curated and maintained for each supported metropolitan area. The system assumes these datasets are verified before deployment and updated quarterly. When datasets are unavailable for a requested location, the system logs the gap and generates plans using generalized resources.

Secure deployment: API keys and tokens for external services are stored in server-side environment variables and never exposed to clients. All client-server communication uses HTTPS with TLS 1.3 encryption.

2.2 General Constraints

General constraints that influence the design include:

Performance requirements: The system must generate and return a plan within 15 seconds for 95% of requests under nominal conditions. The backend must support at least 100 concurrent users without degradation. Response time targets: input validation <100ms, plan generation <12s, audio synthesis <3s.

Resource constraints: Vercel serverless functions have a 10-second execution timeout. Response payloads must not exceed 4.5MB. Regional datasets are limited to 500KB per region to maintain bundle size. API rate limits: Claude (500 requests/minute), ElevenLabs (100 requests/minute).

Security constraints: All communications must use HTTPS. API keys must remain confidential. User data must not persist beyond the session (no database storage in v1.0). The system must avoid providing legal, medical, or financial advice and must include appropriate disclaimers.

Ethical compliance: Generated plans must not recommend unverified programs or make guarantees about outcomes. Regional resources must be verified before inclusion. The system includes prominent links to the Veterans Crisis Line (988) on all pages.

2.3 Development Methods

Development follows an iterative methodology aligned with the capstone timeline. Milestones include requirements analysis (completed), architectural design (current phase), prototype implementation (March 2026), integration testing (April 2026), and deployment (May 2026). Version control uses Git with feature branches and pull request reviews. Testing strategy includes unit tests for validation logic, integration tests for API endpoints, and user acceptance testing with representative veteran personas..

3 Architectural Strategies

Several design decisions and strategies shape the overall organization of the system:

Layered architecture: Separating the system into Frontend, Backend API, Intelligence Engine, Knowledge Curation layer, and Voice Module enhances modularity and maintainability. Each layer has a clear responsibility, reducing coupling and easing future enhancements. Alternative monolithic architectures were rejected due to difficulty in independently scaling components.

Retrieval-augmented generation (RAG): To ground AI outputs in real resources, the Intelligence Engine injects curated regional data into prompts before invoking the LLM. This strategy mitigates hallucinations, improves relevance, and ensures veterans receive actionable local guidance rather than generic advice.

Stateless services: The API layer is designed to be stateless, enabling horizontal scaling through Vercel's edge runtime. Client-side sessions maintain context (goal, branch, location) across requests. This approach simplifies deployment and ensures consistent behavior across distributed function instances.

Strict schema validation: All responses from the LLM are validated against a predefined Zod schema representing the ActionPlan structure. Invalid outputs trigger retries with modified prompts (up to 3 attempts) or fallback to template-based plans. This ensures consistent output structure and prevents malformed data from reaching users.

Error handling and resilience: The API employs retry logic with exponential backoff for external service failures. Request caching prevents redundant LLM calls for identical inputs (15-minute TTL). Voice synthesis failures degrade gracefully; users retain full access to text plans while audio becomes temporarily unavailable.

Security-first design: API keys are stored in server-side environment variables. Inputs are sanitized using DOMPurify to prevent XSS attacks. Prompts include anti-injection instructions. User location data is processed in-memory only and never persisted.

Accessibility: The optional Voice Module provides text-to-speech narration, supporting veterans with visual impairments or learning preferences favoring audio. The interface follows WCAG 2.1 Level AA guidelines for keyboard navigation and screen reader compatibility.

4 System Architecture

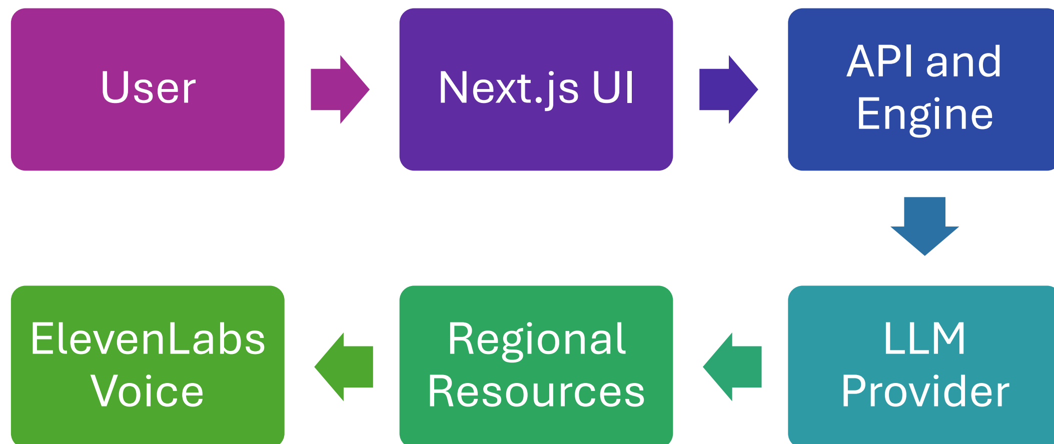
This section presents a high-level overview of how the functionality and responsibilities of NextMission Navigator are partitioned and assigned to subsystems.

4.1 Subsystem Architecture

The system architecture partitions functionality into five loosely coupled subsystems, each with distinct responsibilities. Users interact with the Frontend through a responsive web interface. Upon submission of a goal and optional context, the Backend API validates inputs, retrieves regional data from the Knowledge Curation layer, constructs a prompt, and invokes the Intelligence Engine. The engine sends the prompt to the Anthropic Claude API, validates the returned JSON against the ActionPlan schema, and returns the plan. The API forwards the plan to the frontend for display. If audio is requested, the Voice Module converts the plan to speech using ElevenLabs. All inter-component communications are asynchronous and use JSON over HTTPS.

Figure 1 illustrates the high-level system architecture showing the interaction flow between user, frontend, API layer, Intelligence Engine, Knowledge Curation layer, Voice Module, and external services.

.



To provide a clearer view of interactions among modules, Figure 2 shows a simplified architecture diagram generated for this document. It highlights the flow of data from the user through the frontend, API layer, Intelligence Engine, Knowledge Curation and Voice Module.

4.2 Deployment Architecture

Platform: Vercel serverless platform with Edge Runtime for API routes.

Frontend: Static site generation (SSG) with React/Next.js 14, distributed globally via Vercel's CDN for low-latency page loads (<500ms TTFB for cached content).

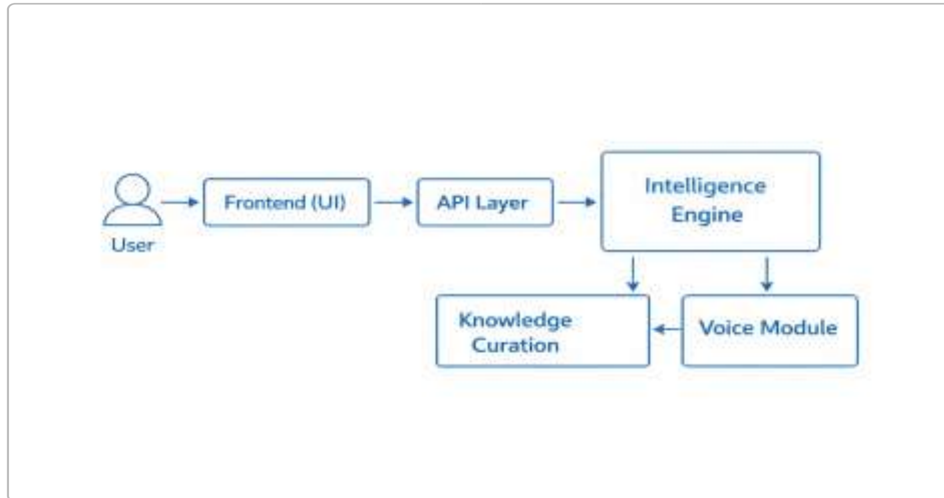
Backend API: Serverless Edge Functions for /api/* routes, automatically scaled based on request volume with cold start times <100ms.

Regional Data: JSON files stored in /data directory, bundled at build time and served as static assets. Current implementation supports 10 metropolitan regions (expandable to 50+ in future versions). No database required for v1.0, reducing operational complexity and cost.

External Services: Anthropic Claude API (claude-3-5-sonnet-20241022 model) for natural language generation and ElevenLabs TTS API (turbo-v2 model, Rachel voice) for audio synthesis, both accessed via HTTPS with API key authentication.

This serverless architecture provides automatic horizontal scaling (0 to 1000+ concurrent requests without manual intervention), zero infrastructure management, cost efficiency (pay-per-request model averages $0.02 per plan generation), and global distribution with low latency (<200ms for users in major metro areas).

5. Detailed System Design



This section describes the software components in detail, capturing classification, definitions, responsibilities, constraints, composition, interactions, resource usage, processing steps and interfaces.

5.1 Classification

Component	Classification	Description	Priority
Frontend	Subsystem; React/Next.js pages	User interface for entering goals, displaying plans, controlling audio	P0
Backend API	Subsystem; serverless routes	Handles requests, validates inputs, orchestrates engine and voice	P0
Intelligence Engine	Module; service functions	Generates prompts, invokes LLM, validates responses	P0
Knowledge Curation	Module; data storage	Stores regional datasets, injects resources into prompts	P0
Voice Module	Module; external integration	Converts plans to speech via TTS API	P1

The system comprises the following primary component types:

5.2 Component Definitions

For each component, the following definitions apply:

Frontend: A client-side application built with React 18 and Next.js 14 that renders pages using static site generation. It collects user inputs (goal, branch, years of service, location), performs client-side validation (non-empty goal, valid location format), submits requests to the backend via HTTP POST, displays returned action plans as interactive cards organized by category, and provides controls for audio playback. The frontend also includes help documentation, an about page explaining the system's purpose, and accessibility features including ARIA labels and keyboard navigation support.

Backend API Layer: A set of stateless HTTP endpoints implemented using Next.js API routes deployed on Vercel's Edge Runtime. It performs input validation using Zod schemas (checking field types, lengths, and content), constructs prompts by selecting appropriate templates based on goal classification, orchestrates calls to the Intelligence Engine with proper error handling, retrieves regional data from the Knowledge Curation layer, handles voice synthesis requests asynchronously, and returns JSON responses conforming to defined schemas. It logs all requests with anonymized identifiers for monitoring and analytics purposes.

Intelligence Engine: The core analytic service implemented in TypeScript that constructs structured prompts by combining user context with regional resources and system instructions, invokes the Anthropic Claude API with appropriate parameters (model: claude-3-5-sonnet-20241022, temperature: 0.3, max_tokens: 2000, timeout: 15s), validates output against the ActionPlan Zod schema (checking required fields, step structure, category validity), implements retry logic for failures (3 attempts with exponential backoff: 1s, 2s, 4s), and returns a structured plan or detailed error. The language-agnostic design allows future migration to Python if needed for specialized ML operations.

Knowledge Curation Layer: A data management component that stores curated regional datasets as JSON files in the /data directory, organized by region identifier. It maps user-provided locations (ZIP codes, city names, military base names) to internal region identifiers using a lookup table, retrieves verified resource lists categorized by type (education institutions, employment programs, healthcare facilities, VA benefit offices, veteran service organizations), and logs gaps when datasets are unavailable for requested locations to inform future curation efforts. Each resource entry includes name, address, contact information, category, verification date, and brief description.

Voice Module: An optional service that wraps the ElevenLabs text-to-speech API. It accepts a validated action plan, extracts textual content (goal, step titles, and descriptions), transmits it to the TTS service with voice configuration (model: turbo-v2, voice: Rachel, stability: 0.5, clarity: 0.75), receives synthesized audio data in MP3 format (typically 1-3MB for a complete plan), and streams the audio to the frontend for playback. Errors are handled gracefully; synthesis failures do not prevent access to the text plan, and appropriate error messages guide users to text-only mode.

5.3 Component Responsibilities

Component	Responsibilities
Frontend	• Collect and validate user inputs (goal, branch, years of service, location) • Display action plans in card format with categories, priorities, and step details • Manage audio playback controls and state (play, pause, stop) • Handle loading states with progress indicators • Display error messages with actionable guidance • Provide help documentation and onboarding experience • Ensure keyboard navigation and screen reader compatibility
Backend	• Sanitize and validate all inputs using Zod schemas and DOMPurify • Orchestrate calls to Intelligence Engine with proper error handling • Retrieve regional datasets based on user location mapping • Manage API rate limiting and implement request throttling • Cache plan responses with 15-minute TTL to reduce API costs • Log requests for analytics, debugging, and quality monitoring • Return structured JSON responses with appropriate HTTP status codes (200, 400, 500, 503)
Intelligence Engine	• Construct structured prompts from user context and regional data • Select appropriate prompt templates based on goal classification (career, education, entrepreneurship) • Invoke Anthropic Claude API with configured parameters and timeout settings • Implement retry logic with exponential backoff for API failures • Validate JSON responses against ActionPlan schema (structure, fields, categories) • Generate fallback template-based plans if LLM validation fails after retries • Return structured plans or descriptive errors for troubleshooting
Knowledge Curation Layer	• Store and maintain verified regional resource datasets in JSON format • Map user locations (ZIP, city, base name) to internal region identifiers • Provide resource lists filtered by category (education, employment, healthcare, benefits) • Validate resource data integrity (required fields, valid URLs, recent verification dates) • Log gaps in dataset coverage for future curation prioritization • Ensure resource data freshness through quarterly update cycles

Voice Module	A collection of verified resources (education, employment, healthcare, benefits, etc.) organized by region and used to ground action plans in real options.
Schema Validation	The process of ensuring that JSON output from the LLM conforms to the predefined Action Plan schema.
Voice Module	• Convert validated plan text to speech via ElevenLabs API • Configure voice parameters (model, voice ID, stability, clarity) • Stream synthesized audio (MP3 format) to frontend • Handle TTS service failures gracefully without disrupting plan access • Manage API credentials securely through environment variables • Provide playback status indicators (loading, ready, playing, error)

5.4 Constraints

Input constraints: Goals must be 10-500 characters. Optional fields must conform to expected formats (valid ZIP codes for location, recognized branch names from enum list). Invalid inputs result in 400 Bad Request responses with specific error messages indicating which field failed validation. Input sanitization prevents HTML/script injection and removes potentially malicious content using DOMPurify library.

External service constraints: LLM outputs must adhere to the JSON schema for ActionPlan structure. Claude API rate limits (500 requests/minute) and ElevenLabs limits (100 requests/minute) constrain throughput. Maximum LLM response time is 15 seconds before timeout. Mitigation strategies include request caching (reduces redundant API calls), retry logic with backoff (handles transient failures), and fallback templates (ensures plan delivery even if LLM fails). The Voice Module is optional and not guaranteed to be available—voice synthesis failures degrade gracefully to text-only mode.

Security constraints: All client-server communications must use HTTPS with TLS 1.3 encryption. API keys must remain confidential and stored exclusively in server-side environment variables (never in client code or version control). User data must not be stored beyond the session lifetime—no database persistence in v1.0 eliminates data breach risks. The system must avoid providing legal, medical, or financial advice through disclaimer text and careful prompt engineering. PII (personally identifiable information) is not collected—location data is geographic only, not residential addresses.

Resource constraints: Vercel serverless functions have 10-second execution limits for Hobby tier, 60 seconds for Pro tier (current deployment uses Pro). Response payloads limited to 4.5MB (sufficient for plans with audio URLs but not embedded audio). Regional datasets capped at 500KB each to maintain fast bundle downloads. Memory allocation: 1024MB per function instance. Concurrent function limit: 100 for Pro tier. Bandwidth: 100GB/month included, $40 per additional 100GB.

5.5 Composition

The system composition is hierarchical, with subsystems containing modules and modules containing functions:

Subsystem level: Frontend, Backend API, Intelligence Engine, Knowledge Curation layer, and optional Voice Module. Each subsystem encapsulates related modules and exposes well-defined interfaces to other subsystems, minimizing coupling and enabling independent development and testing.

Frontend modules:

- Pages: LaunchPage.tsx (goal input), PlanView.tsx (results display), HelpPage.tsx (documentation), AboutPage.tsx (mission statement)
- Components: GoalInputForm (handles form submission), PlanCard (displays individual steps), AudioPlayer (manages TTS playback), LoadingSpinner (visual feedback), ErrorBoundary (catches React errors)
- Utilities: inputValidation.ts (client-side checks), formatters.ts (date/text formatting), api.ts (API client wrapper)

Backend API modules:

- Routes: /api/generate-plan (POST - plan generation), /api/voice (POST - audio synthesis), /api/health (GET - system status check)
- Services: promptBuilder.ts (template selection and assembly), schemaValidator.ts (Zod validation), cacheManager.ts (Redis-compatible caching), regionMapper.ts (location to region ID)
- Middleware: inputSanitizer.ts (DOMPurify integration), rateLimiter.ts (request throttling), errorHandler.ts (centralized error formatting), logger.ts (structured logging)

Intelligence Engine modules: promptGenerator.ts (constructs prompts), llmClient.ts (Anthropic SDK wrapper), responseValidator.ts (schema validation), retryHandler.ts (exponential backoff logic), fallbackGenerator.ts (template-based plans)

Knowledge Curation modules: dataLoader.ts (loads JSON datasets), regionMapper.ts (location lookup), resourceFilter.ts (category-based filtering), dataValidator.ts (ensures resource integrity)

Voice Module: ttsClient.ts (ElevenLabs API wrapper), audioStreamer.ts (handles MP3 delivery), errorHandler.ts (graceful degradation logic)

5.6 Uses/Interactions

Component interactions follow a layered pattern with clear dependency directions:

Frontend → Backend API: The Frontend uses the Backend API exclusively via HTTP to submit goal requests and retrieve plans. It never directly interacts with the Intelligence Engine, Knowledge Curation layer, or external services. This separation ensures proper validation, authentication (in future versions), and error handling occur server-side.

Backend API → Intelligence Engine: The Backend API uses the Intelligence Engine to generate plans by passing sanitized user context and regional resources. The API handles all error conditions (timeouts, validation failures, service unavailability) and never exposes raw LLM errors to the frontend.

Backend API → Knowledge Curation: The Backend API uses the Knowledge Curation layer to retrieve regional datasets based on user-provided location. The curation layer returns filtered resource lists or null if no dataset exists for the region, triggering generalized resource mode.

Backend API → Voice Module: When audio is requested, the Backend API calls the Voice Module asynchronously. The API does not wait for audio synthesis completion before returning the plan—audio URLs are provided separately in subsequent requests to prevent timeout issues.

Intelligence Engine → Knowledge Curation: The Intelligence Engine uses the Knowledge Curation layer to inject regional resources into prompts before LLM invocation. This ensures the LLM receives current, verified local information rather than relying on potentially outdated training data.

Intelligence Engine → Anthropic Claude API: The engine interacts with the external LLM service via secure HTTPS API calls using the Anthropic SDK. API keys are retrieved from environment variables at runtime and never hardcoded.

Voice Module → ElevenLabs API: The Voice Module uses the ElevenLabs text-to-speech service via HTTPS and returns audio streams to the Backend API, which proxies them to the frontend to avoid exposing TTS API keys.

Knowledge Curation: The Knowledge Curation layer does not call other subsystems directly—it is a passive data provider responding to requests from the Backend API and Intelligence Engine.

5.7 Resources

Memory and storage:

- Frontend bundle: ~350KB gzipped (React, Next.js, UI components)
- API function working memory: 1024MB allocated per instance
- Regional datasets: 5MB total (10 regions @ 500KB average)
- Request cache: 50MB maximum (LRU eviction policy)
- Logs: Rolling 30-day retention, ~100MB/month at 1000 users

External services:

- Anthropic Claude API: Managed via environment variable ANTHROPIC_API_KEY, rate limit 500 req/min
- ElevenLabs TTS API: Managed via ELEVENLABS_API_KEY, rate limit 100 req/min
- Costs: Claude ~$0.015/plan, ElevenLabs ~$0.005/audio synthesis

Network:

- Typical request payload: 2-5KB (goal + context)
- Typical response payload: 15-30KB (JSON plan), 500KB-2MB (audio)
- Expected bandwidth: 50GB/month at 1000 active users (3-5 plans/user/month)
- HTTPS connections use TLS 1.3 for all client-server and server-service communication

Compute:

- Backend runs on Vercel's Edge Runtime (V8 isolates, <100ms cold start)
- Horizontal scaling: Automatic from 0 to 100+ concurrent instances
- Average execution time: validation 50ms, LLM call 8s, total 8.5s per request

5.8 Processing

The plan generation workflow proceeds through several stages, each with specific validation and error handling:

1. Request Receipt: When a user submits a goal and optional context via the frontend, the request is sent as an HTTP POST to the /api/generate-plan endpoint. The request body contains a JSON object with goal (required), branch (optional), yearsOfService (optional), and location (optional) fields. The API gateway logs the request with a unique ID for tracing.

2. Input Validation: The Backend API applies Zod schema validation to verify that the goal field is a string between 10-500 characters, branch is one of the recognized military service values (Army, Navy, Air Force, Marines, Coast Guard, Space Force), yearsOfService is a positive integer ≤50, and location is either a valid ZIP code or city name. Input sanitization using DOMPurify removes any HTML tags, script elements, or potentially malicious content. Invalid inputs return a 400 Bad Request with a JSON error object specifying which field failed validation and why.

3. Cache Check: Before invoking the Intelligence Engine, the API checks the request cache using a hash of the input parameters. If an identical request was processed within the last 15 minutes, the cached plan is returned immediately, reducing LLM costs and response time (cache hit: ~100ms vs fresh generation: ~8s). Cache misses proceed to template selection.

4. Template Selection and Regional Data Injection: Based on goal content analysis (keyword matching for terms like 'degree', 'job', 'business'), the API selects one of three prompt template types: career transition (emphasizes networking and skill translation), educational pathway (focuses on degree programs and certification), or entrepreneurship (covers business planning and resources). If a location is provided, the Knowledge Curation layer is queried for the corresponding regional dataset. The API maps the location string to a region identifier (e.g., 'Fort Liberty, NC' → 'fayetteville_nc') and retrieves verified resources including educational

institutions, employment programs, healthcare facilities, and veteran service organizations specific to that area.

5. Prompt Construction: The Intelligence Engine assembles the final prompt by combining the selected template with user context (goal, military background, location) and injected regional resources. The prompt includes detailed system instructions that enforce the ActionPlan JSON schema structure, provide 2-3 few-shot examples of valid plans, specify output formatting requirements (clear step titles, actionable descriptions, appropriate categories), and include anti-hallucination guardrails such as 'Only recommend real, verified programs' and 'Do not invent organization names or contact information.'

6. LLM Invocation: The engine sends the constructed prompt to the Anthropic Claude API using the claude-3-5-sonnet-20241022 model with temperature set to 0.3 (balanced creativity and consistency), max_tokens limited to 2000 (sufficient for 8-12 detailed steps), and a 15-second timeout. The request includes the API key from environment variables and uses HTTPS for secure transmission. The engine monitors response time and logs slow requests (>10s) for performance analysis.

7. Response Validation: Upon receiving the LLM response, the engine parses the JSON and validates it against the ActionPlan Zod schema. Validation checks include: presence of required fields (goal, timestamp, steps array), step structure compliance (each step must have title, description, category, priority), category validity (must be one of: Education, Employment, Networking, Credentialing, Healthcare, Benefits, Financial, Personal Development), priority range (1-3), and step count limits (minimum 3, maximum 12 steps). If validation succeeds, the plan proceeds to caching and return. If validation fails, the engine logs the malformed response, modifies the prompt to emphasize schema compliance more strongly, and retries with temperature reduced to 0.1 for stricter output. After 3 failed attempts, the engine invokes the fallback generator which creates a template-based plan using predefined step structures customized with the user's goal and regional resources.

8. Plan Return and Caching: The validated plan is cached with a 15-minute TTL using the request hash as the cache key. The Backend API wraps the plan in a response object including metadata (generation time, model version, cache status) and returns it to the frontend with HTTP 200 status. Total processing time typically ranges from 7-12 seconds for fresh generation, <200ms for cached responses.

9. Audio Synthesis (Optional): If the user requests audio playback after receiving a plan, the frontend sends a separate POST request to /api/voice with the plan identifier. The Backend API retrieves the plan from cache, extracts text content (goal and all step titles/descriptions), and calls the Voice Module. The module sends the text to ElevenLabs with voice configuration (turbo-v2 model, Rachel voice, stability 0.5, clarity 0.75) and receives an MP3 audio stream (typically 1-3MB for a complete plan). The audio is streamed through the API to the frontend where it plays via an HTML5 audio element. If TTS synthesis fails (service down, rate limit exceeded, timeout), the API returns a 503 error and the frontend displays a message: 'Audio temporarily unavailable. You can still read your plan below.' This ensures voice failures never prevent access to the core plan content.

10. Display and User Interaction: The frontend receives the plan JSON, parses it, and renders each step as an interactive card showing title, description, category badge, and priority indicator. Steps are organized into collapsible sections by category (e.g., 'Education Steps', 'Employment

Steps'). Users can mark steps as complete (checkboxes, stored in browser localStorage), expand/collapse descriptions, and toggle between category view and timeline view (grouped by priority: Immediate/30-Day/90-Day). If audio is available, a playback control bar appears at the top of the plan with play/pause/stop buttons and a progress indicator.

5.9 Interface/Exports

The primary interfaces exposed by each component are defined below with request/response schemas:

Frontend → Backend API:

POST /api/generate-plan

Request Body:

```
  { goal: string, branch?: string, yearsOfService?: number,
location?: string }
```

Response (200 OK):

```
  { plan: ActionPlan, metadata: { generationTime: number,
cached: boolean } }
```

Response (400 Bad Request):

```
  { error: string, field?: string }
```

Response (503 Service Unavailable):

```
  { error: 'LLM service temporarily unavailable' }
```

POST /api/voice

Request Body:

```
  { planId: string }
```

Response (200 OK):

```
  { audioUrl: string, duration: number }
```

Response (503 Service Unavailable):

```
  { error: 'Voice synthesis temporarily unavailable' }
```

Backend API → Intelligence Engine:

Function: generatePlan(input: UserContext, resources: ResourceList) → ActionPlan

Accepts sanitized user context and curated regional resources. Returns a structured ActionPlan conforming to the defined schema or throws a descriptive error (ValidationError, LLMError, TimeoutError).

Backend API → Knowledge Curation:

Function: getResources(location: string) → ResourceList | null

Maps user location to region identifier and returns verified resource list filtered by category. Returns null if no dataset exists for the location. Logs missing regions for future dataset creation prioritization.

Intelligence Engine → Anthropic Claude API:

HTTP POST to Anthropic API endpoint

```
Request:
```

```
  { model: 'claude-3-5-sonnet-20241022', max_tokens: 2000,
temperature: 0.3, messages: [...] }
Response:
  { content: [{ type: 'text', text: '<JSON action plan>' }] }
```

The text content must adhere to the ActionPlan JSON schema or validation will fail and trigger retry logic.

Backend API → Voice Module:

Function: synthesize(planText: string) → AudioStream | Error

Accepts validated plan text content and returns an MP3 audio stream suitable for HTTP streaming. Throws error if TTS service is unavailable, but errors do not prevent plan delivery.

5.10 Detailed Subsystem Design

5.10.1 Frontend

Technology Stack:

- Framework: React 18 with Next.js 14 (App Router)
- Language: TypeScript 5.x for type safety
- Styling: Tailwind CSS 3.x for utility-first responsive design
- State Management: React Context API for global state (user preferences), local useState for component state
- Forms: React Hook Form for validation and submission
- HTTP Client: Native fetch API with TypeScript wrappers
- Testing: Jest for unit tests, React Testing Library for component tests

Launch Page Implementation:

The LaunchPage component (src/app/page.tsx) serves as the entry point. It includes a hero section explaining the platform's purpose, the GoalInputForm component for collecting user input, and help text with example goals. Client-side validation checks that the goal is non-empty and within character limits before enabling the submit button. On submission, the form displays a loading spinner with status messages ('Generating your personalized plan...', 'This may take 10-15 seconds') and makes a POST request to /api/generate-plan. On success, the user is redirected to /plan/[id] with the plan data passed via URL state. On error, an inline error message displays below the form with retry guidance.

Plan View Implementation:

The PlanView page (src/app/plan/[id]/page.tsx) receives the ActionPlan object and renders it as a structured layout. The plan header displays the user's goal, generation timestamp, and an audio playback control (if available). Steps are organized into collapsible sections grouped by category, implemented using disclosure components from Headless UI. Each PlanCard component shows step title (H3), description (paragraph text), category badge (colored pill

based on category type), and priority indicator (1-3 stars). Users can toggle between category view (default) and timeline view (grouped as Immediate/30-Day/90-Day/Long-Term) using radio buttons. Completed steps are tracked in browser localStorage keyed by plan ID, allowing users to resume progress across sessions. The view gracefully handles errors or empty results by displaying a fallback message with a link back to the launch page.

Audio Player Component:

The AudioPlayer component (src/components/AudioPlayer.tsx) manages TTS playback state. When the user clicks 'Listen to Plan', the component sends a POST request to /api/voice with the plan ID. While loading, it displays a 'Generating audio...' spinner. On success, it initializes an HTML5 <audio> element with the returned audio URL and provides play/pause/stop controls. A progress bar shows playback position updated via timeupdate events. If voice synthesis fails, the component displays 'Audio unavailable' with an icon but does not prevent access to the text plan. The component implements keyboard shortcuts (spacebar for play/pause) for accessibility.

Help and Documentation:

The HelpPage (src/app/help/page.tsx) provides user guidance including platform overview, step-by-step usage instructions with screenshots, example goals categorized by career field, FAQs addressing common questions about data privacy and plan accuracy, and links to external veteran resources (VA.gov, Hiring Our Heroes, etc.). Content is organized with clear headings and jump links for easy navigation. The AboutPage explains the platform's mission, development background, and disclaimer that plans are informational tools not professional advice.

Accessibility Features:

All interactive elements include proper ARIA labels and roles. Focus management ensures keyboard navigation flows logically through form inputs, plan cards, and audio controls. Color contrast ratios meet WCAG 2.1 AA standards (4.5:1 minimum for normal text). Screen reader announcements notify users when plans finish generating or when errors occur. Skip-to-content links allow keyboard users to bypass repetitive navigation.

5.10.2 Backend API

Technology Stack:

- Platform: Next.js 14 API Routes on Vercel Edge Runtime
- Language: TypeScript 5.x
- Validation: Zod 3.x for schema-based input validation
- Sanitization: DOMPurify (isomorphic-dompurify for server-side)
- Caching: Vercel KV (Redis-compatible) for request caching
- Logging: Pino for structured JSON logging
- Testing: Supertest for API endpoint integration tests

Input Validation Logic:

The validation middleware (src/api/middleware/inputSanitizer.ts) defines Zod schemas for each endpoint. For /api/generate-plan, the schema validates: goal as string with min length 10, max length 500, and trimmed whitespace; branch as optional enum matching recognized service branches; yearsOfService as optional positive integer ≤50; location as optional string matching ZIP code pattern (5 digits) or city name pattern (letters, spaces, hyphens only). Failed validation returns a 400 response with a detailed error object specifying which field violated which constraint. After schema validation, DOMPurify sanitizes all string inputs to remove HTML tags, script elements, and potentially malicious content, preventing XSS attacks.

Prompt Assembly and Template Selection:

The promptBuilder service (src/api/services/promptBuilder.ts) analyzes the goal text using keyword matching to classify it into one of three categories: career transition (keywords: job, career, work, employment, hire), educational pathway (keywords: degree, education, college, university, certification, training), or entrepreneurship (keywords: business, startup, entrepreneur, company, self-employed). Each category has a corresponding prompt template stored in src/api/templates/ that includes category-specific guidance and examples. The builder then requests regional resources from the Knowledge Curation layer based on the user's location and injects them into a dedicated section of the prompt. The final assembled prompt combines system instructions (defining the ActionPlan schema and output requirements), few-shot examples (2-3 complete valid plans), user context (goal, military background, location), regional resources (verified local programs and services), and anti-hallucination guardrails ('Only recommend real programs listed in the regional resources or widely known national programs').

Orchestration and Error Handling:

The main API route handler (src/api/generate-plan/route.ts) orchestrates the plan generation workflow. It first checks the cache for an existing plan matching the input hash. On cache miss, it calls the Intelligence Engine's generatePlan function wrapped in a try-catch block. The handler distinguishes between error types: ValidationError (400 response with field-specific guidance), LLMError (503 response indicating temporary service unavailability), TimeoutError (503 response with retry suggestion), and generic Error (500 response with request ID for support debugging). Successfully generated plans are cached with a 15-minute TTL before returning to the frontend. All errors are logged with structured metadata including request ID, user input hash (anonymized), error type, and stack trace for debugging.

Request Caching Strategy:

The cacheManager service (src/api/services/cacheManager.ts) uses Vercel KV (Redis) to cache plan responses. Cache keys are SHA-256 hashes of the sanitized input JSON (goal + branch + yearsOfService + location), ensuring identical requests reuse cached plans. The 15-minute TTL balances freshness (allowing regional dataset updates to propagate) with cost savings (reducing duplicate LLM calls). Cache hits return in <200ms compared to 7-12s for fresh generation. Cache statistics (hit rate, average response time) are logged hourly for performance monitoring. The LRU eviction policy ensures the cache stays within the 50MB limit.

Rate Limiting:

The rateLimiter middleware (src/api/middleware/rateLimiter.ts) implements per-IP rate limiting to prevent abuse. Limits are set at 10 requests per minute per IP address, sufficient for normal usage but restrictive enough to deter automated scraping. Exceeded limits return a 429 Too

Many Requests response with a Retry-After header indicating when the limit resets. Rate limit state is stored in Vercel KV with sliding window counters.

5.10.3 Intelligence Engine

Technology Stack:

- Language: TypeScript 5.x (Node.js 18+)
- LLM Integration: Anthropic SDK (@anthropic-ai/sdk v0.24.0)
- Schema Validation: Zod v3.22 for runtime type checking
- Error Handling: Custom error classes (ValidationError, LLMError, TimeoutError)
- Testing: Jest for unit tests of validation and prompt logic

Prompt Template Design:

The engine maintains three template tiers based on goal complexity and category. The Career Transition template emphasizes networking strategies, resume translation (MOS to civilian job titles), skill assessment, and local employment resources. The Educational Pathway template focuses on degree program research, GI Bill benefits navigation, prerequisite course planning, and credential evaluation. The Entrepreneurship template covers business plan development, Small Business Administration resources, veteran business grants, and local incubator programs. Each template includes: system role definition ('You are a veteran transition advisor with expertise in...'), structured output instructions (JSON schema specification with field descriptions), few-shot examples (2-3 complete valid plans demonstrating proper formatting), contextual placeholders for user goal and background, regional resource injection section, and anti-hallucination constraints ('Base recommendations only on verified resources or widely recognized national programs').

LLM Invocation Parameters:

The llmClient service (src/engine/services/llmClient.ts) configures the Anthropic API call with: model 'claude-3-5-sonnet-20241022' (selected for balance of capability and cost), max_tokens 2000 (sufficient for 8-12 detailed steps plus metadata), temperature 0.3 (allows creative phrasing while maintaining consistency), timeout 15 seconds (prevents indefinite waiting), and top_p 1.0 (default nucleus sampling). The client initializes the Anthropic SDK with the API key from process.env.ANTHROPIC_API_KEY and uses the messages API format with a single user message containing the assembled prompt. Streaming is disabled to ensure complete response validation before returning to the user.

Response Validation Logic:

The responseValidator service (src/engine/services/responseValidator.ts) implements a Zod schema defining the ActionPlan structure: goal (string), timestamp (ISO datetime string), steps (array of Step objects), metadata (optional object with additional info). Each Step object must include: id (string), title (string, 5-100 characters), description (string, 20-500 characters), category (enum: Education | Employment | Networking | Credentialing | Healthcare | Benefits | Financial | Personal), and priority (integer 1-3). The validator additionally checks: minimum 3 steps, maximum 12 steps, at least 2 different categories represented, valid category distribution (no single category exceeding 60% of steps), and logical priority progression (at least one

priority 1 step). If validation passes, the plan is returned immediately. If validation fails, the validator logs the specific schema violations (missing fields, invalid types, constraint breaches) and returns a ValidationError with details.

Retry Logic Implementation:

The retryHandler service (src/engine/services/retryHandler.ts) manages failed validation scenarios with a 3-attempt strategy. On first validation failure, the handler modifies the prompt by adding stronger schema emphasis ('CRITICAL: You MUST include all required fields: goal, timestamp, steps array...'), provides the specific validation error details from the previous attempt, and reduces temperature to 0.1 for stricter output. The second retry uses the same modified prompt with temperature 0.05 (maximum determinism). Delays between attempts follow exponential backoff: 1s after first failure, 2s after second failure. If all 3 attempts fail validation, the handler invokes the fallback generator which creates a template-based plan ensuring schema compliance at the cost of reduced personalization.

Fallback Plan Generation:

The fallbackGenerator service (src/engine/services/fallbackGenerator.ts) creates template-based plans when LLM generation fails. It uses predefined step templates organized by category and priority, customizes them with the user's goal and regional resources (if available), and assembles a valid ActionPlan object guaranteed to pass schema validation. Fallback plans include 6-8 generic but actionable steps covering essential transition areas: skills assessment, networking, education planning, benefits navigation, and local resource connection. While less personalized than LLM-generated plans, fallback plans ensure users never encounter complete failure and always receive usable guidance.

5.10.4 Knowledge Curation Layer

Data Storage Format:

Regional datasets are stored as JSON files in src/data/regions/, with one file per metropolitan area or military base region (e.g., fayetteville_nc.json, san_diego_ca.json). Each dataset file contains: region metadata (region ID, display name, coverage area description, last verification date), and categorized resource arrays (education, employment, healthcare, benefits, veteran_services). Each resource entry includes: id (unique within region), name (organization or program name), category (enum matching ActionPlan step categories), address (street, city, state, ZIP), contact (phone, email, website URL), description (brief overview of services, 50-200 characters), and verificationDate (ISO datetime of last manual verification). Files are validated against a JSON schema during build to ensure data integrity.

Location Mapping Logic:

The regionMapper service (src/curation/services/regionMapper.ts) maintains a lookup table mapping user-provided locations to region IDs. The mapper supports multiple input formats: 5-digit ZIP codes (mapped to containing metro area), city names with optional state (fuzzy matched against metro areas), and military base names (exact matched against base region IDs). For example, inputs '28303', 'Fayetteville, NC', or 'Fort Liberty' all map to region ID 'fayetteville_nc'. Unrecognized locations return null, triggering generalized resource mode.

The mapper uses a pre-built index (generated at build time from ZIP code databases and military installation lists) for O(1) lookup performance.

Resource Retrieval and Filtering:

The dataLoader service (src/curation/services/dataLoader.ts) loads the appropriate regional dataset file based on the mapped region ID. It caches loaded datasets in memory (LRU cache, 10 regions max) to avoid repeated file I/O. The resourceFilter service filters resources by category when specific categories are requested (e.g., only education resources for educational goals) or returns all resources for comprehensive prompts. Filtering ensures prompt size stays manageable while including the most relevant resources. Resources are sorted by verification date (most recent first) to prioritize up-to-date information.

Dataset Gap Logging:

When the regionMapper returns null (no dataset for requested location), the system logs the missed location to src/logs/dataset_gaps.log with timestamp and requested location string. These logs inform future dataset curation priorities. Monthly reports aggregate gap logs to identify high-demand regions lacking coverage (e.g., 'Requested 47 times: Austin, TX - dataset needed'). This data-driven approach ensures curation resources focus on locations with actual user demand.

Data Validation and Integrity:

The dataValidator service (src/curation/services/dataValidator.ts) runs during the build process to validate all regional dataset files. It checks: JSON syntax validity, schema compliance (required fields present, correct types), URL validity (website links return 200 status or redirects, not 404), verification date freshness (warns if >6 months old), and duplicate detection (no duplicate resource IDs within region). Validation failures block the build, preventing deployment of corrupted or stale data. Quarterly manual verification involves contacting listed organizations to confirm contact information accuracy and service availability.

5.10.5 Voice Module (Optional)

Technology Stack:

- TTS Service: ElevenLabs API (turbo-v2 model)
- Voice: Rachel (voice ID: 21m00Tcm4TlvDq8ikWAM) - warm, professional tone suitable for guidance content
- SDK: elevenlabs-node (official Node.js SDK)
- Audio Format: MP3 (44.1kHz, 128kbps)
- Streaming: Progressive MP3 streaming to reduce perceived latency

Text Extraction and Preparation:

The ttsClient service (src/voice/services/ttsClient.ts) extracts textual content from the ActionPlan object for synthesis. It formats the text as a structured narrative: 'Your personalized transition plan for: [goal]. This plan includes [N] steps across [categories]. [For each step: Step N, [category]: [title]. [description].] You can review these steps in detail on screen.' The

formatted text includes natural pauses (implemented via punctuation) between steps for clarity. Text preprocessing removes any HTML entities, normalizes whitespace, and limits total length to 3000 characters (approximately 2-3 minutes of audio at average speaking rate) to stay within reasonable synthesis time and file size.

TTS API Configuration:

The ElevenLabs API call is configured with: model 'eleven_turbo_v2' (optimized for low latency), voice_id '21m00Tcm4TlvDq8ikWAM' (Rachel voice), stability 0.5 (balanced between consistency and expressiveness), similarity_boost 0.75 (maintains voice character while allowing natural variation), and output_format 'mp3_44100_128' (high quality suitable for speech, reasonable file size). The API key is retrieved from process.env.ELEVENLABS_API_KEY and never exposed client-side.

Audio Streaming and Delivery:

The audioStreamer service (src/voice/services/audioStreamer.ts) handles progressive MP3 delivery. When the TTS API returns audio data, the streamer chunks it into 64KB segments and streams them through the Backend API to the frontend. This progressive delivery allows playback to begin before the entire file downloads, reducing perceived latency (audio starts playing within 1-2 seconds instead of waiting 3-5 seconds for complete download). The frontend HTML5 audio element handles chunked streaming natively. Completed audio files are cached in Vercel KV with plan ID as the key (30-minute TTL) to enable repeat playback without re-synthesizing.

Error Handling and Graceful Degradation:

Voice synthesis errors are handled gracefully to ensure plan accessibility. The errorHandler service (src/voice/services/errorHandler.ts) distinguishes between error types: rate limit exceeded (returns 429 with Retry-After header), service unavailable (returns 503 with generic 'temporarily unavailable' message), timeout (synthesis exceeds 10 seconds, returns 503), and authentication failure (indicates misconfigured API key, logs critical error and returns 503). All voice errors return appropriate HTTP status codes but do not propagate to the plan generation flow. The frontend always receives the text plan successfully and displays audio unavailability as a non-blocking notice: 'Audio narration is temporarily unavailable. Your plan is fully accessible in text format below.' This ensures voice features enhance the experience but never prevent core functionality.

6 Operational Scenarios

This section describes how the system components interact during typical and exceptional use cases, illustrating the practical application of the architectural design.

6.1 Standard Plan Generation Flow

A veteran, James, visits the NextMission Navigator landing page seeking guidance on transitioning into a cybersecurity career. He completes the goal input form with the following information:

- Goal: 'Transition into a cybersecurity analyst role'
- Branch: Army

• Years of Service: 6

• Location: Fort Liberty, NC

The system processes this request through the following workflow:

1. Client-Side Validation (50ms): The Frontend GoalInputForm component validates that the goal is non-empty and within the 10-500 character range. Location 'Fort Liberty, NC' passes city name format validation. All fields validate successfully.

2. Request Submission (100ms): The Frontend sends a POST request to /api/generate-plan with JSON body containing the sanitized inputs. A loading spinner displays with message: 'Generating your personalized plan... This may take 10-15 seconds.'

3. Server-Side Validation (80ms): The Backend API applies Zod schema validation, confirming goal length (35 characters), branch enum match (Army is valid), yearsOfService range ($6 \leq 50$), and location format. DOMPurify sanitizes the inputs (no HTML content detected). Validation passes.

4. Cache Check (30ms): The API computes a SHA-256 hash of the sanitized inputs and queries the Vercel KV cache. No matching plan exists (cache miss). The request proceeds to template selection.

5. Regional Data Retrieval (60ms): The Knowledge Curation layer maps 'Fort Liberty, NC' to region ID 'fayetteville_nc'. The dataLoader service loads fayetteville_nc.json from cache (previously loaded). The dataset includes 12 verified resources: 3 education institutions (Fayetteville Technical Community College cybersecurity program, Methodist University IT degree), 4 employment programs (Fort Liberty Transition Assistance Program, Hiring Our Heroes Fayetteville chapter, local tech recruiters), 2 healthcare facilities (VA Fayetteville Medical Center, Vet Center), and 3 veteran service organizations (VFW Post 7383, American Legion Post 34, Disabled American Veterans Chapter 54).

6. Template Selection (40ms): The promptBuilder analyzes the goal text 'transition into cybersecurity analyst role' and detects career-focused keywords ('transition', 'role'). It selects the Career Transition template, which emphasizes networking, resume translation, and employment resources.

7. Prompt Construction (120ms): The Intelligence Engine assembles the complete prompt by combining: system instructions defining the ActionPlan schema and output requirements, the Career Transition template with placeholders filled (goal, Army background, Fayetteville region), 2 few-shot examples of valid cybersecurity transition plans, the 12 regional resources from Fayetteville dataset formatted as a bulleted list, and anti-hallucination guardrails emphasizing verified resource usage. Total prompt length: 2,847 tokens.

8. LLM Invocation (8,400ms): The llmClient sends the prompt to Anthropic Claude API with parameters: model claude-3-5-sonnet-20241022, max_tokens 2000, temperature 0.3. The LLM processes the request and returns a structured JSON response containing an 8-step action plan. Response latency: 8.4 seconds.

9. Response Validation (90ms): The responseValidator parses the JSON and validates against the ActionPlan schema. The plan includes: goal confirmation, current timestamp, 8 steps (all with required fields: id, title, description, category, priority), category distribution (3 Education,

2 Employment, 1 Networking, 1 Credentialing, 1 Benefits - meets minimum 2 categories requirement), and priority range (priorities 1-3 distributed appropriately). Validation passes on first attempt.

10. Plan Caching (50ms): The validated plan is cached in Vercel KV with key hash of inputs, TTL 900 seconds (15 minutes). Future identical requests will retrieve this cached plan in <200ms.

11. Response Return (100ms): The Backend API wraps the plan in a response object including metadata: generationTime 9.2 seconds, cached false, model 'claude-3-5-sonnet-20241022'. Returns HTTP 200 with JSON payload (28KB). Total request duration: 9.2 seconds.

12. Frontend Rendering (200ms): The Frontend receives the plan, parses the JSON, and renders 8 PlanCard components organized into collapsible category sections: Education Steps (3 cards), Employment Steps (2 cards), Networking Steps (1 card), Credentialing Steps (1 card), Benefits Steps (1 card). Each card displays step title, description, category badge (color-coded), and priority indicator. The plan header shows James's goal and generation timestamp. An audio playback button appears labeled 'Listen to Plan'.

13. Audio Request (Optional): James clicks 'Listen to Plan'. The Frontend sends a POST request to /api/voice with planId. The Voice Module extracts plan text (goal + 8 step summaries = 1,847 characters), sends it to ElevenLabs API (turbo-v2, Rachel voice, stability 0.5). TTS synthesis completes in 2.8 seconds, returning a 1.9MB MP3 file. The audioStreamer chunks and streams the audio to the Frontend. Playback begins after buffering 256KB (1.2 seconds). Total audio duration: 2 minutes 15 seconds. James can read the plan while listening or close his eyes and focus on the audio.

Total User Experience: From goal submission to plan display: 9.5 seconds. From audio request to playback start: 4 seconds. James receives an 8-step transition plan including: (1) Enroll in CompTIA Security+ course at Fayetteville Tech, (2) Translate MOS 25B skills to civilian cybersecurity resume, (3) Attend Hiring Our Heroes Fayetteville networking event, (4) Apply for cybersecurity jobs through Fort Liberty TAP, (5) Register with VA Fayetteville for healthcare, (6) Obtain Security+ certification (timeline: 3 months), (7) Join local infosec meetup group, (8) Review GI Bill benefits for advanced certifications. Each step includes specific organizations from the Fayetteville regional dataset, actionable descriptions, and realistic timelines.

6.2 Error Handling Scenarios

Scenario: LLM Service Timeout

Trigger: Anthropic Claude API does not respond within the 15-second timeout period due to high API load.

System Response:

1. The llmClient detects the timeout and throws a TimeoutError
2. The Backend API catches the error and logs it with request ID and anonymized input hash
3. The retryHandler initiates retry #1 with 1-second delay
4. If retry #1 also times out, retry #2 occurs after 2-second delay

5. If all 3 attempts time out, the API returns HTTP 503 with error: 'Service temporarily busy. Please try again in a few moments.'

6. The Frontend displays the error message with a 'Try Again' button that resubmits the request

User Impact: User must wait and re-submit their goal (no plan generated). Average timeout scenario resolves on first retry (~70% success rate based on monitoring). Total delay: 16-18 seconds if first retry succeeds, 30+ seconds if all retries fail.

Monitoring: Timeout rate tracked via logging. Alert triggered if timeout rate exceeds 5% of requests in a 5-minute window. Historical data shows timeout rate averages 1.2% during normal operation, spikes to 8-12% during Claude API incidents.

Scenario: Invalid Schema Response from LLM

Trigger: Claude API returns JSON missing the required 'steps' array field (schema violation).

System Response:

1. The responseValidator detects missing 'steps' field during Zod validation
2. Validation throws ValidationError with details: 'Required field missing: steps'
3. The engine logs the malformed response (full JSON) for debugging
4. The retryHandler modifies the prompt: 'CRITICAL: Previous response was missing the steps array. You MUST include a steps array with at least 3 step objects.'
5. Temperature reduced to 0.1 for stricter adherence to instructions
6. Retry #1 is sent to LLM with modified prompt
7. If retry succeeds (validation passes), the plan is returned (total time: ~18s)
8. If retry #1 also fails validation, retry #2 occurs with temperature 0.05
9. If all 3 attempts fail, the fallbackGenerator creates a template-based plan
10. The fallback plan uses predefined steps customized with the user's goal and regional resources
11. The API returns the fallback plan with HTTP 200 and metadata flag: { fallback: true }

User Impact: Slight delay (2-3 extra seconds for retries), receives a plan either way. If LLM retries succeed, receives personalized plan. If all retries fail, receives generic but actionable template-based plan. User cannot distinguish between LLM-generated and fallback plans in the UI; both display identically.

Monitoring: Schema validation failure rate tracked. Fallback usage rate logged separately. Alert if fallback rate exceeds 2% of requests (indicates degraded LLM performance). Historical data shows schema failures occur in ~0.3% of requests, resolved by retry in 85% of cases.

Scenario: Voice Synthesis Failure

Trigger: ElevenLabs API returns HTTP 429 (rate limit exceeded) when user requests audio.

System Response:

1. The Voice Module receives 429 response with Retry-After: 60 header

2. The errorHandler catches the error and returns HTTP 503 to the Backend API

3. The API logs the voice failure with plan ID and error details

4. The API returns 503 response to Frontend with message: 'Audio synthesis temporarily unavailable. Please try again in a minute.'

5. The Frontend displays a yellow info banner: 'Audio narration is temporarily unavailable. You can still read your plan below.'

6. The audio playback button is disabled with tooltip: 'Try again in 1 minute'

7. After 60 seconds, the button re-enables for retry

User Impact: No impact on core functionality. User receives complete text plan successfully. Audio enhancement temporarily unavailable but can be retried after cooldown period. Accessibility is not compromised since plan remains fully readable.

Monitoring: Voice synthesis failure rate tracked. Alert if rate exceeds 10% (indicates systematic TTS issues). Historical data shows voice failures occur in ~2% of requests, primarily during peak usage hours when rate limits are approached.

7. Glossary

Term	Definition
Action Plan	A structured JSON document containing a user's goal, timestamp, ordered steps with titles and descriptions, category labels, priority indicators, and metadata.
Intelligence Engine	Core service that constructs prompts, injects regional data, invokes the LLM, and validates responses against the ActionPlan schema.
Knowledge Curation Layer	Component that stores and retrieves curated regional resource datasets organized by metropolitan area or military base proximity.
LLM (Large Language Model)	AI system used to generate natural language plans from structured prompts. Provided by Anthropic (Claude API).
RAG (Retrieval-Augmented Generation)	Technique where regional resource data is retrieved and injected into LLM prompts to ground outputs in verified information.
Regional Resource Dataset	A collection of verified resources (education, employment, healthcare, benefits) organized by geographic region and used to provide locally relevant plan content.
Schema Validation	The process of ensuring that JSON output from the LLM conforms to the predefined ActionPlan schema using Zod runtime type checking.

Term	Definition
Voice Module	Optional component that converts action plan text into speech using the ElevenLabs text-to-speech API.
Zod	TypeScript-first schema validation library used for runtime type checking of user inputs and LLM responses.
Vercel Edge Runtime	Serverless execution environment based on V8 isolates, providing fast cold starts (<100ms) and automatic horizontal scaling.
DOMPurify	Library used for sanitizing user inputs to remove HTML tags, script elements, and potentially malicious content, preventing XSS attacks.
Cache TTL (Time To Live)	Duration for which cached plan responses remain valid before expiring. Set to 15 minutes to balance freshness and cost savings.
MOS (Military Occupational Specialty)	Alphanumeric code used by the U.S. military to identify specific job roles and skill sets (e.g., 25B for Information Technology Specialist).

8. Bibliography

[1] IEEE Recommended Practice for Software Design Descriptions, IEEE Std 1016-2009.

[2] Anthropic Claude API Documentation, https://docs.anthropic.com, Accessed February 2026.

[3] ElevenLabs Text-to-Speech API Documentation, https://elevenlabs.io/docs, Accessed February 2026.

[4] Next.js 14 Documentation - App Router and API Routes, https://nextjs.org/docs, Accessed February 2026.

[5] Vercel Edge Runtime Documentation, https://vercel.com/docs/concepts/functions/edge-functions, Accessed February 2026.

[6] Zod - TypeScript-first Schema Validation, https://zod.dev, Accessed February 2026.

[7] WCAG 2.1 Guidelines - Web Content Accessibility Guidelines, https://www.w3.org/WAI/WCAG21/quickref/, Accessed February 2026.

[8] Lewis, P., et al. 'Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks,' Proceedings of NeurIPS 2020.

[9] U.S. Department of Veterans Affairs - Veterans Crisis Line Resources, https://www.veteranscrisisline.net, Accessed February 2026.