CUNY AI Schedule Optimizer - Product Requirements Document (PRD)

1. Product Overview & Vision

1.1 Executive Summary
CUNY AI Schedule Optimizer is an AI-powered course planning assistant designed to help CUNY students optimize their academic schedules. The platform integrates real-time course data from CUNYfirst Schedule Builder with AI-driven professor evaluation and schedule conflict detection to provide personalized course recommendations.

1.2 Purpose & Problem Statement
CUNY students face significant challenges when planning their schedules:
- CUNYfirst Schedule Builder lacks intelligent scheduling recommendations
- No centralized platform to compare professor ratings across multiple CUNY schools
- Manual conflict detection is time-consuming and error-prone
- Students lack data-driven guidance on whether specific professor-course combinations are worthwhile
- Difficulty optimizing schedules based on personal preferences (time slots, campus locations, professor quality)

1.3 Vision Statement
Empower CUNY students to build optimal academic schedules by combining intelligent AI-driven recommendations, real-time professor quality insights, and constraint-based scheduling algorithms.

1.4 Target Users
- CUNY undergraduate and graduate students (initial focus)
- Students across all 25 CUNY schools
- Students from all academic disciplines
- Primary demographic: Ages 18-30, tech-savvy, enrolled in 12-18 credits per semester

1.5 Success Metrics & KPIs
- User acquisition: 500+ active CUNY student users within 3 months
- Engagement: 60%+ weekly active usage rate
- NPS (Net Promoter Score): Target 50+
- Schedule optimization: 70%+ of users report improved schedule satisfaction

- Professor rating accuracy: 85%+ alignment with RateMyProfessors aggregate data
- System uptime: 99.5%
- Average response time: <2 seconds for schedule optimization queries
- User retention: 40%+ after 4 weeks

---

2. Product Scope & Features

2.1 Core Features (MVP - Phase 1)

2.1.1 Schedule Optimization Engine
- **Intelligent Schedule Suggestions**: AI analyzes user preferences and generates optimal class schedules
  - Input: Required courses, preferred time slots, campus preferences, max/min class hours per day
  - Output: 3-5 optimized schedule recommendations ranked by preference score
  - Algorithm: Constraint satisfaction solver with Gemini AI refinement
  - Response time: < 3 seconds

2.1.2 Professor Evaluation System
- **AI-Powered Professor Grades**: Analyze aggregated RateMyProfessors reviews to assign A-F grades
  - Input: Professor name, subject, university
  - Output: Letter grade (A-F), composite score (0-100), aspect breakdowns
  - Aspects analyzed: Teaching clarity, grading fairness, engagement, accessibility
  - Data freshness: Updated weekly

2.1.3 Schedule Conflict Detection
- **Automatic Conflict Prevention**: Identify and prevent course time conflicts
  - Detect overlapping class times
  - Identify unrealistic travel times between campuses
  - Flag scheduling constraints (prerequisites, corequisites)
  - Suggest alternative sections in real-time

2.1.4 Course Search & Discovery

- **Advanced Course Search**:
  - Filter by: Department, course code, credits, time slot, professor, campus location, course modality
  - Sort by: Professor rating, class size, times offered
  - Display: Course description, prerequisites, enrollment cap, current enrollment

2.1.5 Professor Comparison Tool
- **Side-by-side Professor Comparison**: Compare 2-4 professors teaching the same course
  - Show ratings, difficulty scores, review highlights, key feedback themes
  - Recommend best professor-course combinations
  - Historical enrollment trends

2.2 Phase 2 Features (Post-MVP)

2.2.1 Personalization Engine
- **Learning Style Matching**: Adapt professor recommendations based on student learning style
  - Learning style assessment quiz
  - Match with professor teaching methods from reviews
  - Personalized confidence scores

2.2.2 Real-Time Monitoring
- **Schedule Availability Alerts**: Notify when preferred courses become available
  - Push notifications when sections open/close
  - Alert when new professor sections added
  - Early registration suggestions

2.2.3 Degree Audit Integration
- **DegreeWorks Integration** (if API available): Pull required courses automatically
  - Map required courses to available sections
  - Suggest course sequences
  - Compliance checking

2.2.4 Schedule Sharing & Collaboration
- **Share & Get Feedback**: Share proposed schedules with peers/advisors

```
  - Generate shareable schedule links
  - Collect feedback from advisors
  - Compare schedules with classmates

---

## 3. Functional Requirements

### 3.1 User Authentication & Profiles
- **FR-1.1**: System shall authenticate users via CUNY SSO (LDAP/CUNYfirst integration)
- **FR-1.2**: Users can create local email-based backup accounts
- **FR-1.3**: User profiles shall store: major, graduation year, preferences, saved schedules
- **FR-1.4**: Data persistence: All user data encrypted at rest in Supabase PostgreSQL

### 3.2 Course Data Management
- **FR-2.1**: System shall sync course data from CUNYfirst API weekly
- **FR-2.2**: For each course section, store: course ID, name, credits, professor, time slots, room, campus, enrollment cap, modality
- **FR-2.3**: System shall handle courses across all CUNY schools (25 institutions)
- **FR-2.4**: Data validation: Flag courses with missing data, prevent stale data serving

### 3.3 Professor Rating Pipeline
- **FR-3.1**: System shall scrape RateMyProfessors data weekly for all CUNY professors
- **FR-3.2**: System shall cache RateMyProfessors ratings with timestamps
- **FR-3.3**: Sentiment analysis: Extract and classify 5 sentiment aspects from each review
  - Teaching clarity, grading fairness, engagement, accessibility, class difficulty
- **FR-3.4**: Scoring formula: Weighted average of aspects with recency decay
- **FR-3.5**: System shall generate professor "grades" A-F with < 5 minute latency

### 3.4 AI Schedule Optimization
```

- **FR-4.1**: System shall accept user preferences as input (required courses, time constraints, campus preference)
- **FR-4.2**: System shall generate 3-5 optimized schedules within 3 seconds
- **FR-4.3**: Each schedule shall be ranked with a preference score (0-100)
- **FR-4.4**: System shall validate all generated schedules for conflicts
- **FR-4.5**: Optimization criteria: Minimize travel time, maximize professor ratings, respect user time preferences, balance course load

3.5 Conflict Detection
- **FR-5.1**: System shall detect overlapping class times (same section cannot have overlaps)
- **FR-5.2**: System shall identify unrealistic travel times between campuses (< 10 min walking assumed)
- **FR-5.3**: System shall check prerequisites/corequisites
- **FR-5.4**: Real-time validation: Conflicts detected and reported before schedule submission

3.6 User Interface
- **FR-6.1**: Dashboard shall display: current schedule, optimization suggestions, upcoming classes
- **FR-6.2**: Schedule Builder: Drag-and-drop course selection, visual timetable
- **FR-6.3**: Professor cards: Show rating, grade, top reviews, difficulty level
- **FR-6.4**: Responsive design: Works on desktop (primary), tablet, mobile devices
- **FR-6.5**: Accessibility: WCAG 2.1 AA compliance

---

4. Non-Functional Requirements

4.1 Performance
- **NFR-1.1**: Schedule optimization response time: < 3 seconds (p95)
- **NFR-1.2**: Course search results: < 500ms
- **NFR-1.3**: Professor rating lookup: < 1 second (from cache)
- **NFR-1.4**: Page load time: < 2 seconds (Largest Contentful Paint)
- **NFR-1.5**: API response time: < 100ms for all endpoints

4.2 Reliability & Availability
- **NFR-2.1**: System uptime: 99.5% (monthly)
- **NFR-2.2**: Graceful degradation: Service available even if RateMyProfessors scraper fails
- **NFR-2.3**: Database backup: Daily automated backups, 30-day retention
- **NFR-2.4**: Error handling: Comprehensive error messages, automatic error logging

4.3 Scalability
- **NFR-3.1**: System shall support 10,000+ concurrent users
- **NFR-3.2**: Database connection pooling: Max 100 concurrent connections
- **NFR-3.3**: Caching strategy: Redis-like caching for professor ratings, course search results
- **NFR-3.4**: Horizontal scaling: Load balanced across multiple instances

4.4 Security
- **NFR-4.1**: All data encryption: TLS 1.3 in transit, AES-256 at rest
- **NFR-4.2**: Authentication: OAuth 2.0 via Supabase
- **NFR-4.3**: Authorization: Row-Level Security (RLS) in Supabase PostgreSQL
- **NFR-4.4**: Rate limiting: 100 requests/minute per user, 10,000 requests/hour per IP
- **NFR-4.5**: GDPR compliance: Data privacy policy, user data deletion capability
- **NFR-4.6**: Input validation: Prevent SQL injection, XSS attacks
- **NFR-4.7**: Secrets management: Environment variables, no hardcoded credentials

4.5 Maintainability
- **NFR-5.1**: Code documentation: README, inline comments, API documentation (OpenAPI)
- **NFR-5.2**: Logging: Structured JSON logging, centralized log aggregation
- **NFR-5.3**: Monitoring: Real-time dashboards for API performance, database health, error rates
- **NFR-5.4**: CI/CD: Automated testing, linting, deployment pipeline

4.6 Compliance & Legal

- **NFR-6.1**: Terms of Service: Clear usage restrictions, data privacy policy
- **NFR-6.2**: RateMyProfessors: Respect robots.txt, rate limiting on scraper (1 request/second)
- **NFR-6.3**: CUNY Data: No storage of sensitive CUNY data beyond course catalog
- **NFR-6.4**: Accessibility: WCAG 2.1 AA standards

---

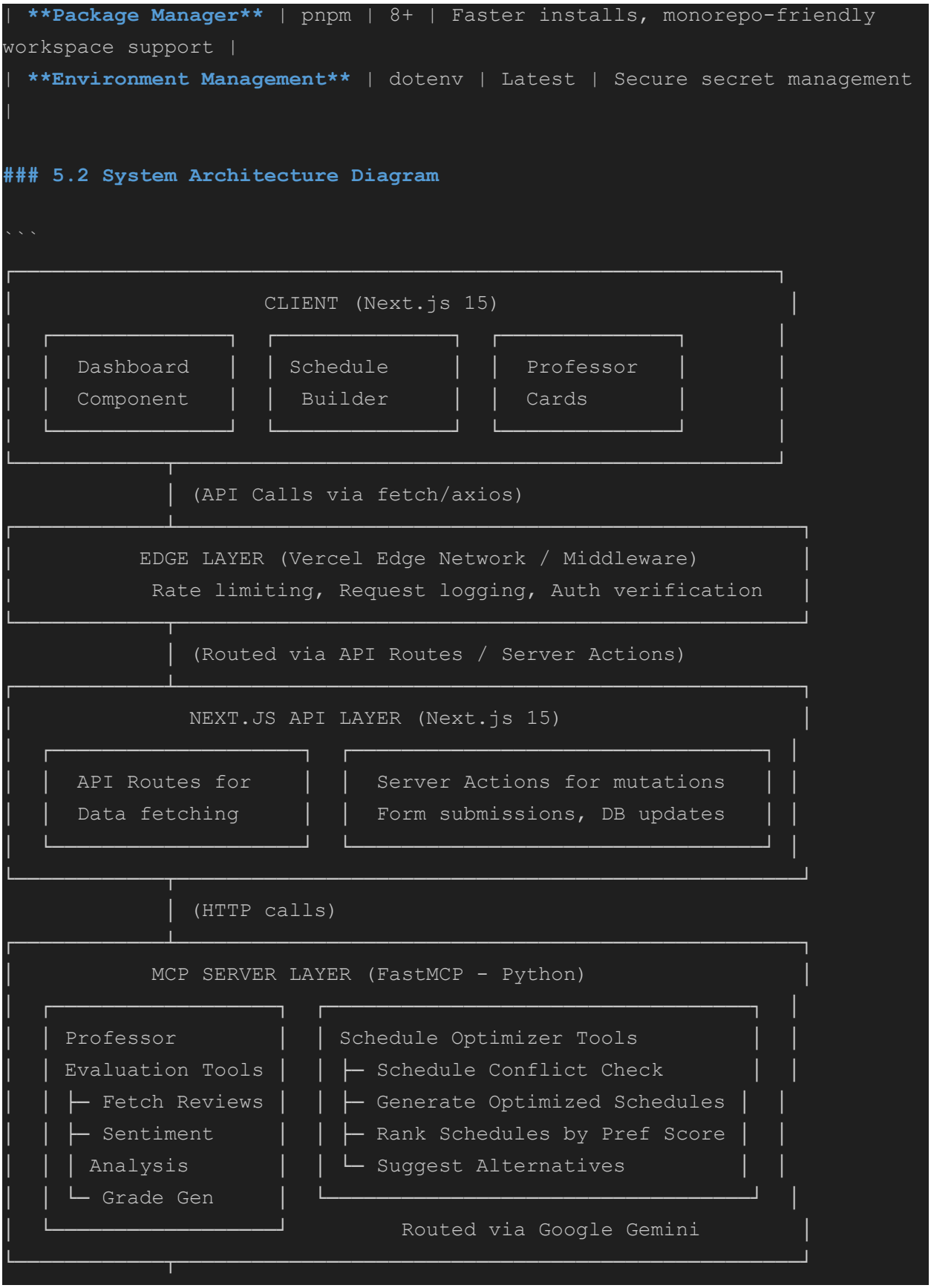
5. Technical Architecture

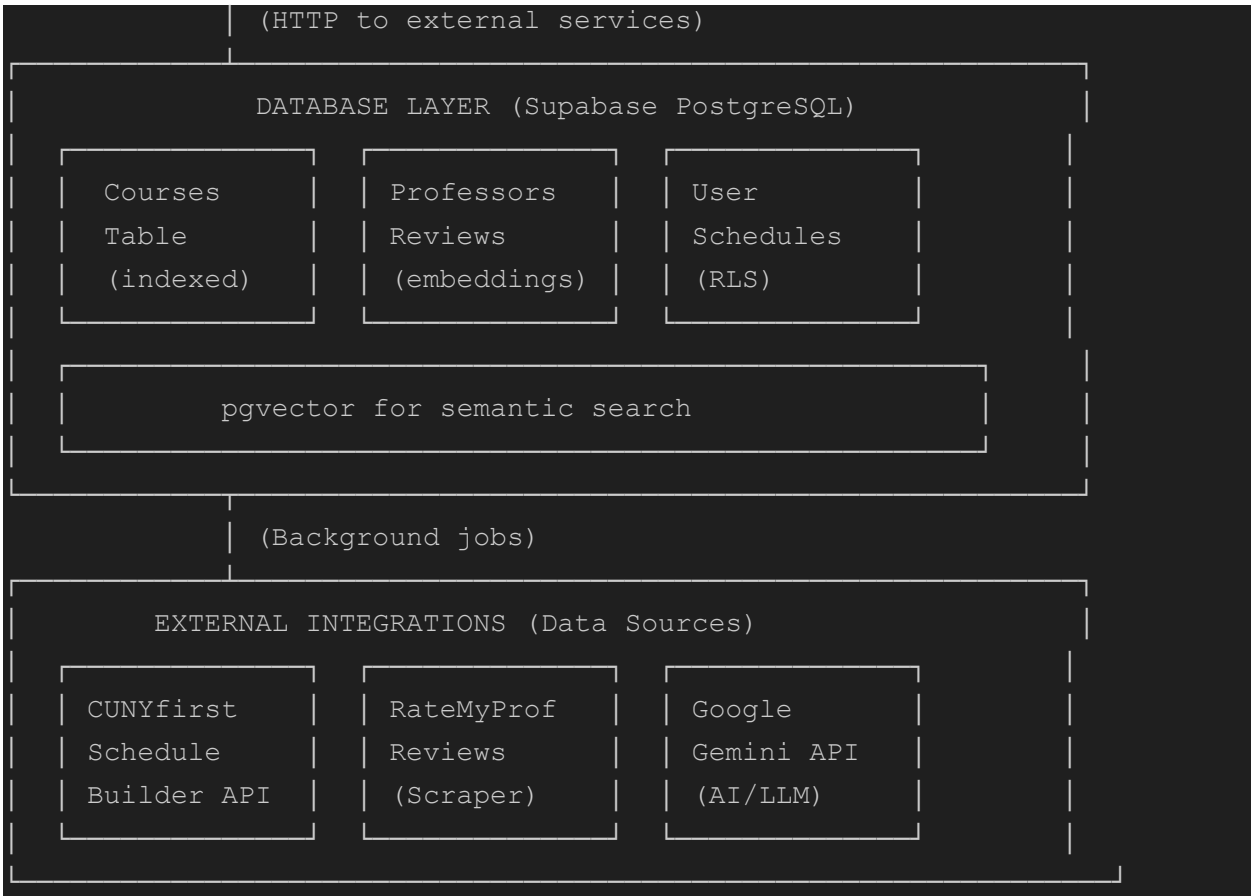
5.1 Technology Stack

Component	Technology	Version	Rationale
Frontend	Next.js 15	15.x	React Server Components, App Router, built-in optimizations, Vercel deployment
Frontend Auth	Supabase Auth	Latest	OAuth 2.0, CUNY SSO integration, built-in MFA support
Backend API/Middleware	Next.js API Routes / Server Actions	15.x	Minimal overhead, integrates with frontend, easy deployment
AI/LLM Integration	Google Gemini API	2.0-flash	Lower cost than OpenAI, excellent function calling, multimodal support
MCP Framework	FastMCP (Python)	Latest	Easy tool definition, async support, Gemini-compatible
Database	Supabase PostgreSQL	14+	pgvector for embeddings, real-time subscriptions, built-in auth
Vector Store	pgvector (PostgreSQL)	Latest	Semantic search for courses, stored in same DB
Caching	Supabase (edge caching)	N/A	Cache layer in Vercel Edge Network
Sentiment Analysis	Transformers (Hugging Face)	Latest	Fine-tuned models for reviews, deployed in Python backend
Data Scraping	BeautifulSoup4 + Selenium	Latest	RateMyProfessors data collection
Task Scheduling	APScheduler (Python)	3.10+	Weekly data sync, scheduled updates
Deployment	Vercel (Frontend) + Railway/Fly.io (Backend)	Latest	Seamless Next.js deployment, Python backend support

| **Package Manager** | pnpm | 8+ | Faster installs, monorepo-friendly workspace support |
| **Environment Management** | dotenv | Latest | Secure secret management |

5.2 System Architecture Diagram





5.3 Data Flow: Schedule Optimization

```
User Input
    ↓
    | (Required courses, preferences)
    ↓
Next.js Server Action
    ↓
    | Fetch course sections from Supabase
    ├─ Validate against conflicts
    └─ Get cached professor ratings
    ↓
Call Gemini API with function tools
    ↓
    | Gemini decides to call MCP tools
    ├─ fetch_course_sections()
```

```
    ├─ check_scheduling_constraints()
    ├─ get_professor_grades()
    └─ generate_optimized_schedule()
    ↓
MCP Server (FastMCP - Python)
    ├─ Constraint satisfaction solver
    ├─ Sentiment analysis on reviews
    └─ Schedule optimization algorithm
    ↓
Return structured JSON results
    ↓
Gemini AI ranks & formats results
    ↓
Next.js returns to client
    ↓
React renders schedule recommendations
```

---

6. Repository Structure & Monorepo Organization

6.1 Monorepo vs Polyrepo Decision: MONOREPO ✅

Rationale for Monorepo:
- Shared types between frontend and backend (course structures, professor ratings)
- Atomic commits across frontend/backend changes
- Easier dependency management (both use Python and JavaScript)
- Simpler deployment process
- Easier to maintain consistency and version alignment
- Smaller team (solo developer initially) benefits from centralized codebase
- CI/CD pipeline runs all tests in one workflow

Tools: pnpm workspaces (frontend) + Python poetry (backend)

6.2 Directory Structure

```
cuny-schedule-optimizer/                    # Root monorepo
│
├── 📄 README.md                            # Project overview
├── 📄 CONTRIBUTING.md                      # Development guidelines
├── 📄 pnpm-workspace.yaml                  # pnpm workspace config
├── 📄 package.json                         # Root package.json (shared
scripts)
├── 📄 pyproject.toml                       # Python project config
├── 📄 .env.example                         # Example environment variables
├── 📄 turbo.json                           # Turborepo build config
(optional)
│
├── 📁 apps/                                # Next.js applications
│   ├── web/                                # Main student-facing app
│   │   ├── 📄 package.json
│   │   ├── 📄 tsconfig.json
│   │   ├── 📄 next.config.ts
│   │   ├── 📄 tailwind.config.ts
│   │   ├── 📁 app/                         # App Router (Next.js 15)
│   │   │   ├── layout.tsx                  # Root layout with auth check
│   │   │   ├── page.tsx                    # Home page
│   │   │   ├── (auth)/                     # Auth route group
│   │   │   │   ├── login/page.tsx
│   │   │   │   ├── signup/page.tsx
│   │   │   │   └── callback/page.tsx       # OAuth callback
│   │   │   ├── (dashboard)/                # Protected routes
│   │   │   │   ├── dashboard/page.tsx      # Main dashboard
│   │   │   │   ├── schedule-builder/page.tsx
│   │   │   │   ├── professor-comparison/page.tsx
│   │   │   │   └── my-schedules/page.tsx
│   │   │   ├── api/                        # API routes
│   │   │   │   ├── optimize-schedule/route.ts
│   │   │   │   ├── get-courses/route.ts
│   │   │   │   ├── get-professor-grade/route.ts
│   │   │   │   ├── save-schedule/route.ts
│   │   │   │   └── health/route.ts
│   │   │   └── error.tsx                   # Error boundary
│   │   ├── 📁 components/
```

```
│   │   │   ├── Schedule/
│   │   │   │   ├── ScheduleGrid.tsx       # Visual schedule
│   │   │   │   ├── ScheduleConflictAlert.tsx
│   │   │   │   └── CourseCard.tsx
│   │   │   ├── Professor/
│   │   │   │   ├── ProfessorCard.tsx
│   │   │   │   ├── ProfessorComparison.tsx
│   │   │   │   └── GradeDisplay.tsx
│   │   │   ├── Search/
│   │   │   │   ├── CourseSearch.tsx
│   │   │   │   ├── FilterPanel.tsx
│   │   │   │   └── SearchResults.tsx
│   │   │   ├── Common/
│   │   │   │   ├── Header.tsx
│   │   │   │   ├── Navigation.tsx
│   │   │   │   ├── LoadingSpinner.tsx
│   │   │   │   └── ErrorBoundary.tsx
│   │   │   └── Layout/
│   │   │       ├── Sidebar.tsx
│   │   │       └── MainLayout.tsx
│   │   ├── 📁 hooks/
│   │   │   ├── useAuth.ts                 # Auth context hook
│   │   │   ├── useScheduleOptimization.ts # Schedule API hook
│   │   │   ├── useCourseSearch.ts         # Course search hook
│   │   │   └── useProfessorData.ts        # Professor ratings hook
│   │   ├── 📁 lib/
│   │   │   ├── api.ts                     # API client functions
│   │   │   ├── supabase.ts                # Supabase client config
│   │   │   ├── auth.ts                    # Auth utilities
│   │   │   ├── types.ts                   # Shared TypeScript types
│   │   │   └── utils/
│   │   │       ├── formatTime.ts
│   │   │       ├── detectConflicts.ts
│   │   │       └── calculateScore.ts
│   │   ├── 📁 styles/
│   │   │   ├── globals.css
│   │   │   └── variables.css
│   │   └── 📁 public/
│   │       └── images/
│   │
```

```
│   └── admin/                          # (Future) Admin dashboard
│       └── ... (Similar structure to web)
│
├── 📁 packages/                        # Shared packages
│   ├── shared-types/                   # TypeScript type definitions
│   │   ├── 📄 package.json
│   │   ├── 📄 tsconfig.json
│   │   ├── index.ts
│   │   ├── course.types.ts             # Course/Section types
│   │   ├── professor.types.ts          # Professor rating types
│   │   ├── schedule.types.ts           # Schedule types
│   │   └── user.types.ts               # User types
│   │
│   ├── ui-components/                  # Reusable UI components
│   │   ├── 📄 package.json
│   │   ├── 📄 tsconfig.json
│   │   ├── index.ts
│   │   ├── Button/
│   │   │   ├── Button.tsx
│   │   │   └── Button.test.tsx
│   │   ├── Card/
│   │   └── Modal/
│   │
│   └── database/                       # Shared DB schemas & migrations
│       ├── 📄 package.json
│       ├── migrations/
│       │   ├── 001_init_schema.sql
│       │   ├── 002_add_professors.sql
│       │   ├── 003_add_user_schedules.sql
│       │   ├── 004_add_reviews.sql
│       │   └── 005_add_embeddings.sql
│       └── seeds/
│           └── seed_cuny_schools.sql
│
├── 📁 services/                        # Python backend services
│   ├── 📄 pyproject.toml
│   ├── 📄 poetry.lock
│   ├── 📄 README.md
│   ├── 📁 mcp_server/                  # FastMCP server
│   │   ├── 📄 __init__.py
```

```
│   │   ├── main.py                        # MCP server entry point
│   │   ├── config.py                      # Environment config
│   │   ├── tools/
│   │   │   ├── __init__.py
│   │   │   ├── professor_evaluator.py     # Professor rating tools
│   │   │   │   ├── fetch_professor_reviews()
│   │   │   │   ├── analyze_review_sentiment()
│   │   │   │   └── generate_professor_grade()
│   │   │   ├── schedule_optimizer.py      # Schedule optimization tools
│   │   │   │   ├── fetch_course_sections()
│   │   │   │   ├── check_scheduling_constraints()
│   │   │   │   ├── detect_conflicts()
│   │   │   │   └── generate_optimized_schedule()
│   │   │   └── helper.py                  # Shared utilities
│   │   ├── models/
│   │   │   ├── __init__.py
│   │   │   ├── course.py
│   │   │   ├── professor.py
│   │   │   └── schedule.py
│   │   ├── services/
│   │   │   ├── __init__.py
│   │   │   ├── supabase_service.py        # Database operations
│   │   │   ├── ratemyprof_scraper.py      # RateMyProf scraping
│   │   │   ├── sentiment_analyzer.py      # NLP sentiment analysis
│   │   │   ├── constraint_solver.py       # Schedule optimization
│   │   │   └── gemini_client.py           # Gemini API calls
│   │   ├── utils/
│   │   │   ├── __init__.py
│   │   │   ├── logger.py                  # Structured logging
│   │   │   ├── cache.py                   # Caching utilities
│   │   │   └── validators.py              # Input validation
│   │   └── tests/
│   │       ├── __init__.py
│   │       ├── test_professor_evaluator.py
│   │       ├── test_schedule_optimizer.py
│   │       └── fixtures/
│   │
│   ├── 📁 background_jobs/                # Scheduled tasks
│   │   ├── 📄 __init__.py
│   │   ├── scheduler.py                   # APScheduler setup
```

```
│   │   ├── jobs/
│   │   │   ├── __init__.py
│   │   │   ├── sync_courses.py            # Weekly course sync
│   │   │   ├── scrape_reviews.py          # Weekly review scraping
│   │   │   ├── analyze_reviews.py         # Weekly sentiment analysis
│   │   │   └── update_professor_grades.py # Weekly grade update
│   │   └── logs/
│   │
│   └── requirements/
│       ├── base.txt                       # Base dependencies
│       ├── dev.txt                        # Development dependencies
│       └── prod.txt                       # Production dependencies
│
├── 📁 docs/                               # Documentation
│   ├── API.md                             # API documentation
│   ├── ARCHITECTURE.md                    # System architecture
│   ├── DATABASE.md                        # Database schema
│   ├── DEPLOYMENT.md                      # Deployment guide
│   ├── DEVELOPMENT.md                     # Development setup
│   └── MCP_TOOLS.md                       # MCP tools reference
│
├── 📁 .github/                            # GitHub configuration
│   ├── workflows/
│   │   ├── ci.yml                         # CI pipeline
│   │   ├── test.yml                       # Test pipeline
│   │   ├── deploy-frontend.yml            # Vercel deployment
│   │   └── deploy-backend.yml             # Python backend deployment
│   └── ISSUE_TEMPLATE/
│
├── 📁 scripts/                            # Utility scripts
│   ├── setup-dev.sh                       # Development setup
│   ├── seed-db.sh                         # Database seeding
│   ├── run-tests.sh                       # Run all tests
│   └── deploy.sh                          # Deployment script
│
├── 📄 .env.example                        # Environment template
├── 📄 .gitignore
├── 📄 .dockerignore
├── 📄 docker-compose.yml                  # Local dev environment
└── 📄 Makefile                            # Common commands
```

```

---

## 7. Google Gemini API Integration

### 7.1 Why Google Gemini over OpenAI

| Criteria | Gemini 2.0 | GPT-4o |
|---|---|---|
| **Cost** | $0.075/1M input tokens, $0.30/1M output tokens | $5/1M input, $15/1M output | ✅ |
| **Function Calling** | Excellent, async support | Excellent | ⚖️ |
| **Multimodality** | Images, audio, video | Images, audio | ✅ |
| **Response Time** | ~500-800ms | ~600-900ms | ✅ |
| **Context Window** | 1M tokens | 128K tokens | ✅ |
| **MCP Support** | Native in Gemini CLI | Via LangChain | ✅ |
| **Reliability** | High | High | ⚖️ |

### 7.2 Gemini Integration Architecture

```python
# services/mcp_server/services/gemini_client.py
import os
from google import genai
from fastmcp import Client as MCPClient
from fastmcp.client.transports import FastMCPTransport

class GeminiScheduleOptimizer:
    def __init__(self, mcp_server):
        self.client = genai.Client(api_key=os.getenv("GEMINI_API_KEY"))
        self.model = "gemini-2.0-flash"
        self.transport = FastMCPTransport(mcp_server)
        self.mcp_client = MCPClient({"transport": self.transport})

    async def optimize_schedule(self, user_input: dict) -> dict:
        """
        Use Gemini with MCP tools to optimize schedule
        """
```
```

```
        # Build prompt with user constraints
        system_prompt = """You are a CUNY academic advisor specializing in
schedule optimization.

        Available tools:
        - fetch_course_sections(): Get available course sections
        - check_scheduling_constraints(): Validate course compatibility
        - detect_conflicts(): Find schedule conflicts
        - get_professor_grades(): Get professor quality ratings
        - generate_optimized_schedule(): Create optimal schedules

        For each request:
        1. Understand user's required courses and preferences
        2. Fetch available sections
        3. Check all constraints
        4. Generate 3-5 optimized schedules
        5. Rank by student preference
        6. Explain your reasoning
        """

        user_message = f"""Please optimize my CUNY schedule:
        - Required courses: {user_input['courses']}
        - Preferred times: {user_input['time_preferences']}
        - Campus: {user_input['campus']}
        - Avoid gaps > 2 hours: {user_input['avoid_gaps']}

        Generate the best 3 schedules with detailed explanations.
        """

        response = await self.client.aio.models.generate_content(
            model=self.model,
            contents=[
                {"role": "user", "parts": [{"text": user_message}]}
            ],
            system_instruction=system_prompt,
            tools=[self._get_tools()]  # MCP tools converted to Gemini
format
        )

        return self._parse_response(response)
```

```
    def _get_tools(self):
        """Convert MCP tools to Gemini function calling format"""
        return [
            {
                "type": "function",
                "function": {
                    "name": "fetch_course_sections",
                    "description": "Get available course sections with
times and professors",
                    "parameters": {
                        "type": "object",
                        "properties": {
                            "course_codes": {"type": "array", "items":
{"type": "string"}},
                            "semester": {"type": "string"}
                        },
                        "required": ["course_codes"]
                    }
                }
            },
            # ... other tools
        ]
```

7.3 Gemini API Key Setup
- Create key at: https://aistudio.google.com/apikey
- Set `GEMINI_API_KEY` in environment variables
- Rate limit: 15 requests per minute (free tier), upgrade as needed

---

8. Development & Deployment Plan

8.1 Development Environment Setup

Prerequisites:
- Node.js 20+
- Python 3.11+
- PostgreSQL (or use Supabase)

- pnpm (package manager)
- Git

Setup Steps:
```bash
# Clone and install dependencies
git clone <repo>
cd cuny-schedule-optimizer
pnpm install

# Install Python dependencies
cd services
poetry install
cd ..

# Copy environment template
cp .env.example .env.local
# Fill in:
# - GEMINI_API_KEY
# - SUPABASE_URL
# - SUPABASE_SERVICE_ROLE_KEY
# - DATABASE_URL

# Run development servers
pnpm dev        # Starts Next.js on 3000
pnpm dev:backend # Starts FastMCP server on 8000
```

8.2 Deployment Architecture

Frontend (Vercel):
- Automatic deployment from `main` branch
- Environment variables synced from Supabase project
- CDN for static assets
- Serverless functions for API routes

Backend (Railway or Fly.io):
- Python FastAPI/FastMCP service
- Background jobs (APScheduler)
- PostgreSQL connection pooling

```
- Automated restarts on failure

**Database (Supabase)**:
- Managed PostgreSQL
- Automatic backups
- Real-time subscriptions
- pgvector for semantic search

### 8.3 CI/CD Pipeline

```yaml
# .github/workflows/ci.yml
on: [push, pull_request]

jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - uses: actions/setup-node@v3
        with:
          node-version: 20
      - uses: actions/setup-python@v4
        with:
          python-version: "3.11"
      - name: Install dependencies
        run: pnpm install && cd services && poetry install
      - name: Run tests
        run: pnpm test && pnpm test:backend
      - name: Linting
        run: pnpm lint
      - name: Type checking
        run: pnpm type-check

  deploy-frontend:
    needs: test
    if: github.ref == 'refs/heads/main'
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
```
```

```
      - uses: vercel/action@main
        with:
          vercel-token: ${{ secrets.VERCEL_TOKEN }}
          vercel-org-id: ${{ secrets.VERCEL_ORG_ID }}
          vercel-project-id: ${{ secrets.VERCEL_PROJECT_ID }}

  deploy-backend:
    needs: test
    if: github.ref == 'refs/heads/main'
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - name: Deploy to Railway/Fly
        # Configure based on chosen platform
```

---

9. Database Schema Overview

9.1 Core Tables

```sql
-- Users table
CREATE TABLE users (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  email TEXT UNIQUE NOT NULL,
  name TEXT,
  major TEXT,
  graduation_year INT,
  preferences JSONB DEFAULT '{}',
  created_at TIMESTAMP DEFAULT NOW(),
  updated_at TIMESTAMP DEFAULT NOW()
);

-- Courses table
CREATE TABLE courses (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  course_code TEXT NOT NULL,  -- e.g., "CSC381"
  subject_code TEXT,          -- e.g., "CSC"
```

```
  course_number TEXT,          -- e.g., "381"
  name TEXT NOT NULL,
  description TEXT,
  credits INT,
  department TEXT,
  university TEXT,             -- CUNY school
  created_at TIMESTAMP DEFAULT NOW(),
  UNIQUE(course_code, university)
);

-- Course sections table
CREATE TABLE course_sections (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  course_id UUID REFERENCES courses(id) ON DELETE CASCADE,
  section_number TEXT,
  professor_id UUID,
  days TEXT,                   -- "MWF", "TTh", etc.
  start_time TIME,
  end_time TIME,
  room TEXT,
  campus TEXT,
  modality TEXT,               -- In-person, online, hybrid
  capacity INT,
  enrolled INT,
  created_at TIMESTAMP DEFAULT NOW()
);

-- Professors table
CREATE TABLE professors (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  name TEXT NOT NULL,
  ratemyprof_id TEXT,
  university TEXT,
  department TEXT,
  average_rating NUMERIC(3,2),
  average_difficulty NUMERIC(3,2),
  review_count INT,
  grade_letter CHAR(1),        -- A-F
  composite_score NUMERIC(3,0),
  last_updated TIMESTAMP,
```

```
  created_at TIMESTAMP DEFAULT NOW()
);

-- Professor reviews table
CREATE TABLE professor_reviews (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  professor_id UUID REFERENCES professors(id) ON DELETE CASCADE,
  review_text TEXT,
  rating NUMERIC(3,2),
  difficulty NUMERIC(3,2),
  sentiment_positive INT,
  sentiment_aspects JSONB,   -- {clarity: 0.85, fairness: 0.72, ...}
  created_at TIMESTAMP DEFAULT NOW(),
  scraped_at TIMESTAMP
);

-- User schedules table
CREATE TABLE user_schedules (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  user_id UUID REFERENCES users(id) ON DELETE CASCADE,
  semester TEXT,             -- "Fall 2025", "Spring 2025"
  name TEXT DEFAULT 'My Schedule',
  sections UUID[],           -- Array of section IDs
  created_at TIMESTAMP DEFAULT NOW(),
  updated_at TIMESTAMP DEFAULT NOW()
);

-- Create indexes for performance
CREATE INDEX idx_courses_code ON courses(course_code);
CREATE INDEX idx_sections_professor ON course_sections(professor_id);
CREATE INDEX idx_professors_name ON professors(name);
CREATE INDEX idx_reviews_professor ON professor_reviews(professor_id);
CREATE INDEX idx_schedules_user ON user_schedules(user_id);
```

9.2 pgvector Setup (Semantic Search)

```sql
CREATE EXTENSION IF NOT EXISTS vector;
```

```
-- Add embedding column for course descriptions
ALTER TABLE courses ADD COLUMN description_embedding vector(1536);

-- Create index for semantic search
CREATE INDEX ON courses USING ivfflat (description_embedding
vector_cosine_ops);

-- Query example: Find similar courses
SELECT * FROM courses
WHERE description_embedding <->
  $1::vector < 0.5
ORDER BY description_embedding <-> $1::vector
LIMIT 10;
```

---

10. Milestones & Timeline

Phase 1: MVP (Weeks 1-4)
- **Week 1**: Setup monorepo, Supabase, Vercel deployment
- **Week 2**: Frontend: Dashboard, course search, basic schedule builder
- **Week 3**: Backend: Course sync, professor scraping, sentiment analysis
- **Week 4**: Integration: Gemini API + MCP tools, testing, deployment

Phase 2: Enhancement (Weeks 5-8)
- Real-time monitoring & alerts
- Learning style matching
- Schedule sharing

Phase 3: Scale (Weeks 9+)
- DegreeWorks integration
- Advanced analytics
- Mobile app

---

11. Success Criteria

- ✅ First 50 CUNY students sign up within 2 weeks of launch

- ✅ 70% of users report improved schedule satisfaction
- ✅ Professor grade accuracy >85% vs. RateMyProf aggregate
- ✅ System response time <3 seconds for schedule optimization
- ✅ Zero critical bugs in first month
- ✅ NPS score >40

---

12. Risk Mitigation

Risk	Impact	Mitigation
RateMyProf scraping blocked	High	Implement API fallback, cache aggressively, implement respectful scraping
Gemini API rate limit	Medium	Implement queuing, cache LLM responses, upgrade tier if needed
Database performance	High	Implement indexing, connection pooling, query optimization, monitoring
Student adoption	High	Beta test with 10+ students, iterate based on feedback
Course data sync failure	High	Implement retry logic, fallback to cached data, alerts

---

13. Success Metrics & Analytics

Key Metrics to Track
- Daily/weekly active users (DAU/WAU)
- Schedule optimization requests per day
- Professor rating lookups
- User retention (1 week, 1 month)
- System uptime and error rate
- API latency (p50, p95, p99)
- Cost per user (infrastructure)

Analytics Integration
- Implement event tracking (Mixpanel, Segment, or Google Analytics 4)
- Monitor via Vercel Analytics Dashboard
- Set up error tracking (Sentry)

- Database query performance monitoring

CUNY AI Schedule Optimizer - Implementation Guide: CUNY Global Search Integration

Quick Reference: Updated Data Source

Primary Data Source: CUNY Global Search Website (instead of CUNYfirst Schedule Builder API)
- URL: https://globalsearch.cuny.edu/
- Method: Web scraping (with ethical considerations)
- Update frequency: Weekly (with option for real-time if possible)
- Coverage: All 25 CUNY schools

---

1. CUNY Global Search Integration Strategy

1.1 Data Extraction Architecture

```
CUNY Global Search Website
         |
         ├─ Search Query (Course code, semester)
         |
         ├─ HTML Response
         |       ├─ Course listings
         |       ├─ Section details
         |       ├─ Professor names
         |       ├─ Time slots
         |       ├─ Campus locations
         |       └─ Enrollment info
         |
         ├─ Parsing Pipeline
         |       ├─ BeautifulSoup (HTML parsing)
         |       ├─ Regex (data extraction)
         |       ├─ Data validation
```

```
          │       └─ Normalization
          │
          ├─ Storage
          │       └─ Supabase PostgreSQL
          │
          └─ Cache Layer
              └─ Update weekly
```

1.2 Scraper Implementation

Technology:
- **BeautifulSoup4** for HTML parsing
- **Selenium** (headless Chrome) for JavaScript-rendered content
- **httpx** for async HTTP requests
- **APScheduler** for weekly scheduling

File: `services/mcp_server/services/cuny_global_search_scraper.py`

```python
import asyncio
from bs4 import BeautifulSoup
import httpx
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import WebDriverWait
import logging

logger = logging.getLogger(__name__)

class CUNYGlobalSearchScraper:
    BASE_URL = "https://globalsearch.cuny.edu/"

    def __init__(self):
        self.session = None
        self.headers = {
            'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36'
        }
```

```python
    async def scrape_semester_courses(self, semester: str) -> list:
        """
        Scrape all courses for a given semester from CUNY Global Search
        Semester format: "Fall 2025", "Spring 2026", etc.
        """
        courses = []

        # Get list of all CUNY schools
        cuny_schools = await self._get_cuny_schools()

        for school in cuny_schools:
            logger.info(f"Scraping {school['name']} for {semester}")
            try:
                school_courses = await self._scrape_school_courses(
                    school_code=school['code'],
                    school_name=school['name'],
                    semester=semester
                )
                courses.extend(school_courses)
            except Exception as e:
                logger.error(f"Error scraping {school['name']}: {e}")
                continue

        return courses

    async def _scrape_school_courses(self, school_code: str, school_name:
str, semester: str) -> list:
        """
        Scrape courses for a specific school and semester
        """
        courses = []

        # CUNY Global Search uses pagination and dynamic loading
        # We'll use Selenium for JavaScript rendering
        driver = webdriver.Chrome(options=self._get_chrome_options())

        try:
            # Construct search URL for the school
            search_url =
f"{self.BASE_URL}?school={school_code}&semester={semester}"
```

```
            driver.get(search_url)

            # Wait for results to load (JavaScript rendering)
            WebDriverWait(driver, 10).until(
                lambda d: d.find_elements(By.CLASS_NAME, "course-listing")
            )

            # Get the rendered HTML
            html = driver.page_source
            soup = BeautifulSoup(html, 'html.parser')

            # Parse course listings
            course_elements = soup.find_all('div',
class_='course-listing')

            for element in course_elements:
                try:
                    course_data = self._parse_course_element(element,
school_name, semester)
                    if course_data:
                        courses.append(course_data)
                except Exception as e:
                    logger.warning(f"Error parsing course element: {e}")
                    continue

        finally:
            driver.quit()

        return courses

    def _parse_course_element(self, element, school_name: str, semester:
str) -> dict:
        """
        Extract course details from a course listing element
        Expected structure may vary; adjust based on actual HTML
        """
        try:
            # Example parsing (adjust based on actual CUNY Global Search
HTML structure)
```

```
            course_code = element.find('span',
class_='course-code')?.text.strip()
            course_name = element.find('span',
class_='course-name')?.text.strip()
            credits = element.find('span', class_='credits')?.text.strip()

            # Get all section details
            sections = []
            section_elements = element.find_all('tr', class_='section')

            for section_el in section_elements:
                section_data = self._parse_section_element(section_el)
                sections.append(section_data)

            return {
                'course_code': course_code,
                'course_name': course_name,
                'credits': int(credits) if credits else None,
                'university': school_name,
                'semester': semester,
                'sections': sections,
                'scraped_at': datetime.now()
            }

        except Exception as e:
            logger.error(f"Error parsing course element: {e}")
            return None

    def _parse_section_element(self, element) -> dict:
        """
        Extract section details (professor, time, location, etc.)
        """
        try:
            section_number = element.find('td',
class_='section-num')?.text.strip()
            professor_name = element.find('td',
class_='professor')?.text.strip()
            time_range = element.find('td', class_='time')?.text.strip()
# "MWF 10:00-11:15 AM"
            location = element.find('td', class_='location')?.text.strip()
```

```
            modality = element.find('td', class_='modality')?.text.strip()
# "In-person", "Online"
            enrollment = element.find('td',
class_='enrollment')?.text.strip()  # "45/50"

            # Parse time slots
            days, start_time, end_time = self._parse_time_slot(time_range)

            # Parse enrollment
            enrolled, capacity = self._parse_enrollment(enrollment)

            return {
                'section_number': section_number,
                'professor_name': professor_name,
                'days': days,
                'start_time': start_time,
                'end_time': end_time,
                'location': location,
                'modality': modality,
                'enrolled': enrolled,
                'capacity': capacity
            }

        except Exception as e:
            logger.error(f"Error parsing section element: {e}")
            return None

    def _parse_time_slot(self, time_str: str) -> tuple:
        """
        Parse time slot like "MWF 10:00-11:15 AM" or "Tu 6:30-8:15 PM"
        Returns: (days: str, start_time: time, end_time: time)
        """
        import re
        from datetime import datetime, time

        if not time_str:
            return None, None, None

        # Pattern: "MWF 10:00-11:15 AM" or "Online"
        if 'Online' in time_str or 'TBA' in time_str:
```

```
            return 'Online', None, None

        # Extract days and times
        match =
re.match(r'([A-Z]+)\s+(\d{1,2}):(\d{2})-(\d{1,2}):(\d{2})\s+(AM|PM)',
time_str)

        if match:
            days, start_h, start_m, end_h, end_m, period = match.groups()

            # Convert 12-hour to 24-hour format
            if period == 'PM' and start_h != '12':
                start_h = int(start_h) + 12
            elif period == 'AM' and start_h == '12':
                start_h = 0

            start_time = time(int(start_h), int(start_m))
            end_time = time(int(end_h), int(end_m))

            return days, start_time, end_time

        return None, None, None

    def _parse_enrollment(self, enrollment_str: str) -> tuple:
        """
        Parse enrollment like "45/50"
        Returns: (enrolled: int, capacity: int)
        """
        if not enrollment_str:
            return None, None

        match = re.match(r'(\d+)/(\d+)', enrollment_str)
        if match:
            return int(match.group(1)), int(match.group(2))

        return None, None

    async def _get_cuny_schools(self) -> list:
        """
        Get list of all CUNY schools with their codes
```

```
        Can be hardcoded or fetched from CUNY Global Search
        """
        return [
            {'code': 'CCNY', 'name': 'City College'},
            {'code': 'HUNTER', 'name': 'Hunter College'},
            {'code': 'QC', 'name': 'Queens College'},
            {'code': 'BARUCH', 'name': 'Baruch College'},
            {'code': 'BROOKLYN', 'name': 'Brooklyn College'},
            {'code': 'LEHMAN', 'name': 'Lehman College'},
            {'code': 'YORK', 'name': 'York College'},
            {'code': 'CSI', 'name': 'College of Staten Island'},
            {'code': 'KBCC', 'name': 'Kingsborough Community College'},
            {'code': 'LAGUARDIA', 'name': 'LaGuardia Community College'},
            {'code': 'BMCC', 'name': 'Borough of Manhattan Community
College'},
            {'code': 'NYCCC', 'name': 'NYC College of Technology'},
            {'code': 'HOSTOS', 'name': 'Hostos Community College'},
            {'code': 'MEDGAR', 'name': 'Medgar Evers College'},
            {'code': 'GC', 'name': 'Graduate School'},
            # ... add all 25 schools
        ]

    def _get_chrome_options(self):
        """Configure Selenium Chrome options"""
        from selenium.webdriver.chrome.options import Options

        options = Options()
        options.add_argument('--headless')
        options.add_argument('--no-sandbox')
        options.add_argument('--disable-dev-shm-usage')
        options.add_argument(f'user-agent={self.headers["User-Agent"]}')

        return options
```

---

2. Database Storage Strategy

2.1 Updated Schema for CUNY Global Search Data

```sql
-- Courses table (synced from CUNY Global Search)
CREATE TABLE courses (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  course_code TEXT NOT NULL,           -- e.g., "CSC381"
  subject_code TEXT,                   -- e.g., "CSC"
  course_number TEXT,                  -- e.g., "381"
  name TEXT NOT NULL,
  credits INT,
  university TEXT NOT NULL,            -- CUNY school name
  semester TEXT NOT NULL,              -- "Fall 2025"
  description TEXT,
  last_scraped TIMESTAMP DEFAULT NOW(),
  UNIQUE(course_code, university, semester)
);

-- Course sections (from CUNY Global Search)
CREATE TABLE course_sections (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  course_id UUID REFERENCES courses(id) ON DELETE CASCADE,
  section_number TEXT NOT NULL,
  professor_id UUID REFERENCES professors(id) ON DELETE SET NULL,
  professor_name TEXT,                 -- Cache professor name if not in
DB yet

  -- Schedule details
  days TEXT,                           -- "MWF", "TTh", "Online", etc.
  start_time TIME,
  end_time TIME,

  -- Location
  location TEXT,                       -- Room number/building
  modality TEXT,                       -- "In-person", "Online", "Hybrid"

  -- Enrollment
  enrolled INT,
  capacity INT,

  -- Metadata
```

```
  scraped_at TIMESTAMP DEFAULT NOW(),
  updated_at TIMESTAMP DEFAULT NOW(),

  UNIQUE(course_id, section_number, semester)
);

-- Index for fast lookups
CREATE INDEX idx_courses_semester ON courses(semester);
CREATE INDEX idx_courses_university ON courses(university);
CREATE INDEX idx_sections_professor ON course_sections(professor_name);
CREATE INDEX idx_sections_time ON course_sections(days, start_time);
```

2.2 Data Sync Pipeline (APScheduler)

File: `services/background_jobs/jobs/sync_cuny_courses.py`

```python
from apscheduler.schedulers.asyncio import AsyncIOScheduler
from datetime import datetime, timedelta
from cuny_global_search_scraper import CUNYGlobalSearchScraper
from supabase_service import SupabaseService

class CUNYCourseSyncJob:
    def __init__(self):
        self.scraper = CUNYGlobalSearchScraper()
        self.db = SupabaseService()
        self.scheduler = AsyncIOScheduler()

    async def sync_all_semesters(self):
        """Sync courses for current and next semesters"""
        semesters = self._get_semesters_to_sync()

        for semester in semesters:
            logger.info(f"Starting sync for {semester}")

            try:
                # Scrape all courses from CUNY Global Search
                courses = await
self.scraper.scrape_semester_courses(semester)
```

```
                # Store in database
                await self.db.insert_courses(courses)

                # Update sync timestamp
                await self.db.update_sync_timestamp(semester)

                logger.info(f"Successfully synced {len(courses)} courses
for {semester}")

            except Exception as e:
                logger.error(f"Error syncing {semester}: {e}")
                # Send alert notification
                await self._send_error_alert(semester, str(e))

    def _get_semesters_to_sync(self) -> list:
        """Determine which semesters to sync"""
        current_month = datetime.now().month
        current_year = datetime.now().year

        semesters = []

        # Determine current semester
        if current_month < 6:
            semesters.append(f"Spring {current_year}")
            semesters.append(f"Summer {current_year}")
            semesters.append(f"Fall {current_year}")
        elif current_month < 8:
            semesters.append(f"Summer {current_year}")
            semesters.append(f"Fall {current_year}")
        else:
            semesters.append(f"Fall {current_year}")
            semesters.append(f"Spring {current_year + 1}")

        return semesters

    def start_scheduler(self):
        """Start background scheduler"""
        # Sync every Sunday at 2 AM (low traffic time)
        self.scheduler.add_job(
```

```
            self.sync_all_semesters,
            'cron',
            day_of_week='sun',
            hour=2,
            minute=0
        )
        self.scheduler.start()

# Initialize and start
sync_job = CUNYCourseSyncJob()
sync_job.start_scheduler()
```

---

3. Handling CUNY Global Search Limitations

3.1 Technical Challenges & Solutions

Challenge	Impact	Solution
Dynamic JavaScript Content	Data not visible in raw HTML	Use Selenium for rendering, handle async loads
Rate Limiting	IP may be blocked	Implement exponential backoff, rotate user agents, respect robots.txt
Page Structure Changes	Scraper breaks with site updates	Use flexible CSS selectors, error handling, monitoring
Large Dataset	Scraping all 25 schools takes time	Parallelize scraping, cache aggressively, incremental updates
Missing Professor IDs	Can't match to RateMyProf	Implement fuzzy matching, manual curation, user corrections
Incomplete Course Data	Some sections missing details	Use defaults, fallback data, request user input in UI

3.2 Retry Logic & Error Handling

```python
import asyncio
from tenacity import retry, stop_after_attempt, wait_exponential
```

```
class RobustCUNYScraper(CUNYGlobalSearchScraper):
    @retry(
        stop=stop_after_attempt(3),
        wait=wait_exponential(multiplier=1, min=2, max=10)
    )
    async def _scrape_school_courses_with_retry(self, **kwargs):
        """Retry scraping with exponential backoff"""
        return await self._scrape_school_courses(**kwargs)

    async def scrape_with_fallback(self, semester: str):
        """Try scraping, fallback to cached data if fails"""
        try:
            courses = await self.scrape_semester_courses(semester)
            return courses
        except Exception as e:
            logger.error(f"Scraping failed: {e}. Using cached data.")

            # Fallback to previous semester's data (same courses likely
offered)
            last_year_semester = f"{semester.split()[0]}
{int(semester.split()[1]) - 1}"
            cached_courses = await
self.db.get_courses_by_semester(last_year_semester)

            # Update semester label but keep other data
            for course in cached_courses:
                course['semester'] = semester

            return cached_courses
```

---

4. Rate Limiting & Ethical Scraping

4.1 Best Practices

```python
import time
import random
```

```
class EthicalCUNYScraper(CUNYGlobalSearchScraper):
    def __init__(self, request_delay: float = 2.0):
        super().__init__()
        self.request_delay = request_delay
        self.last_request_time = 0

    async def _make_request(self, url: str):
        """Rate-limited requests with respect for server resources"""
        # Respect minimum delay between requests
        elapsed = time.time() - self.last_request_time
        if elapsed < self.request_delay:
            await asyncio.sleep(self.request_delay - elapsed)

        # Random delay to avoid bot detection (1-3 seconds)
        delay = random.uniform(1, 3)
        await asyncio.sleep(delay)

        # Make request
        async with httpx.AsyncClient() as client:
            response = await client.get(url, headers=self.headers,
timeout=10)
            self.last_request_time = time.time()

            return response

    def _respect_robots_txt(self):
        """Check and respect robots.txt from CUNY Global Search"""
        try:
            response = httpx.get(f"{self.BASE_URL}/robots.txt")
            # Parse robots.txt and verify we're allowed to scrape
            logger.info(f"robots.txt:\n{response.text}")
        except Exception as e:
            logger.warning(f"Could not fetch robots.txt: {e}")
```

4.2 Alternative: API Access

Best Approach: Request CUNY for official API access
- Contact: Office of Information Technology (OIT) at CUNY

- Benefit: Official support, no risk of IP blocking, official data governance
- Documentation: Inquire about CUNYfirst API or Schedule Builder API

---

5. Integration with Gemini MCP Tools

5.1 Updated MCP Tool: Fetch Courses from CUNY Global Search

```python
# services/mcp_server/tools/schedule_optimizer.py

@mcp.tool
async def fetch_course_sections(
    course_codes: list[str],
    semester: str = None
) -> dict:
    """
    Fetch available course sections from CUNY Global Search data

    Args:
        course_codes: List of course codes (e.g., ["CSC381", "CSC382"])
        semester: Semester (e.g., "Fall 2025"). Defaults to current semester.

    Returns:
        Dictionary with course sections, professors, times, and enrollment
    """
    if not semester:
        semester = get_current_semester()

    try:
        sections = []

        for course_code in course_codes:
            # Query Supabase for courses matching code in current semester
            response = supabase.from_('courses').select(
                'id, course_code, name, credits, university'
```

```
            ).eq('course_code', course_code).eq('semester',
semester).execute()

            if not response.data:
                return {
                    'error': f'Course {course_code} not found for
{semester}',
                    'data': []
                }

            course_id = response.data[0]['id']

            # Get all sections for this course
            sections_response = supabase.from_('course_sections').select(
                'id, section_number, professor_name, days, start_time,
end_time, '
                'location, modality, enrolled, capacity'
            ).eq('course_id', course_id).execute()

            sections.extend(sections_response.data)

        return {
            'success': True,
            'semester': semester,
            'total_sections': len(sections),
            'sections': sections
        }

    except Exception as e:
        logger.error(f"Error fetching sections: {e}")
        return {'error': str(e), 'data': []}


@mcp.tool
async def get_professor_grades(
    professor_names: list[str],
    course_code: str = None
) -> dict:
    """
    Get AI-generated grades for professors based on RateMyProf reviews
```

```
    Args:
        professor_names: List of professor names
        course_code: Optional course code to filter ratings

    Returns:
        Dictionary with professor grades and metrics
    """
    results = []

    for prof_name in professor_names:
        # Query professor with grades
        query = supabase.from_('professors').select(
            'id, name, grade_letter, composite_score, average_rating, '
            'average_difficulty, review_count'
        ).eq('name', prof_name)

        if course_code:
            query = query.eq('subject_code', course_code.split()[0])

        response = query.execute()

        if response.data:
            results.append(response.data[0])
        else:
            results.append({
                'name': prof_name,
                'grade': 'N/A',
                'error': 'No ratings found'
            })

    return {
        'professors': results,
        'timestamp': datetime.now().isoformat()
    }
```

---

6. UI Integration: Show Data Source

6.1 Update UI to Display CUNY Global Search Source

```typescript
// app/components/CourseSearch/CourseSearch.tsx

export function CourseSearch() {
  return (
    <div className="course-search">
      <h2>Search CUNY Courses</h2>
      <p className="data-source">
        📊 Data sourced from{' '}
        <a
          href="https://globalsearch.cuny.edu/"
          target="_blank"
          rel="noopener noreferrer"
        >
          CUNY Global Search
        </a>
        {' '}• Last updated: {lastUpdateTime} • Next sync: Sunday 2 AM EST
      </p>

      <SearchForm />
      <SearchResults />
    </div>
  )
}
```

---

7. Updated Implementation Checklist

Phase 1: MVP (Updated with CUNY Global Search)

- [ ] **Week 1**:
  - [ ] Setup monorepo, Supabase
  - [ ] Research CUNY Global Search structure
  - [ ] Build web scraper with Selenium

```
- [ ] **Week 2**:
  - [ ] Test scraper on 2-3 CUNY schools
  - [ ] Store course data in Supabase
  - [ ] Frontend: Dashboard, course search

- [ ] **Week 3**:
  - [ ] Scrape all 25 CUNY schools
  - [ ] Setup background sync job (APScheduler)
  - [ ] Professor scraping + sentiment analysis

- [ ] **Week 4**:
  - [ ] Gemini API + MCP integration
  - [ ] Schedule optimization with real CUNY data
  - [ ] Testing & deployment

---

## 8. Monitoring & Alerts

### 8.1 Key Metrics to Monitor

```python
# services/mcp_server/utils/monitoring.py

class ScraperMonitoring:
    """Monitor CUNY Global Search scraper health"""

    async def log_sync_metrics(self, semester: str, metrics: dict):
        """
        Log sync metrics to database
        metrics: {courses_scraped, sections_scraped, errors,
duration_seconds}
        """
        await supabase.from_('sync_logs').insert({
            'semester': semester,
            'courses_scraped': metrics['courses_scraped'],
            'sections_scraped': metrics['sections_scraped'],
            'errors': metrics['errors'],
            'duration_seconds': metrics['duration_seconds'],
            'timestamp': datetime.now()
```
```

```
        }).execute()

    async def check_data_staleness(self):
        """Alert if course data is too old"""
        last_sync = await self._get_last_sync_time()
        hours_since_sync = (datetime.now() - last_sync).total_seconds() /
3600

        if hours_since_sync > 168:  # 7 days
            await self._send_alert(
                f"CUNY course data is {hours_since_sync:.1f} hours old. "
                f"Sync may have failed."
            )
```

---

9. Cost Estimate (Updated)

Component	Cost	Notes
Vercel (Frontend)	Free/$25/mo	Hobby or Pro
Supabase (Database)	$25/mo	Pro tier for production
Google Gemini API	$50-200/mo	Based on usage (~1M tokens/day)
Railway/Fly.io (Backend)	$5-20/mo	Python scraper + scheduler
Domain	$15/yr	Custom domain
Total	**$95-275/mo**	Scales with user growth

---

10. Testing Against CUNY Global Search

10.1 Test Cases

```python
# services/mcp_server/tests/test_cuny_scraper.py

import pytest

@pytest.mark.asyncio
```

```python
async def test_scrape_single_school():
    """Test scraping one CUNY school"""
    scraper = CUNYGlobalSearchScraper()
    courses = await scraper._scrape_school_courses(
        school_code='CCNY',
        school_name='City College',
        semester='Fall 2025'
    )

    assert len(courses) > 0
    assert courses[0]['university'] == 'City College'
    assert 'sections' in courses[0]

@pytest.mark.asyncio
async def test_parse_section_time():
    """Test time parsing"""
    scraper = CUNYGlobalSearchScraper()
    days, start, end = scraper._parse_time_slot("MWF 10:00-11:15 AM")

    assert days == "MWF"
    assert start.hour == 10
    assert end.hour == 11

@pytest.mark.asyncio
async def test_database_sync():
    """Test storing scraped data in Supabase"""
    # ... test database insertion
```