

AI Prompts Used - Spectral Analyzer Hackathon

1. need to scaffold a PyQt6 desktop app for spectral analysis. Structure should include: Main window w/ modern UI, drag-drop file handling for CSV files, preview panel for data viz, settings dialog for API config, resource mgmt for icons and styles. can you create the basic project structure with init.py files, main.py entry point, and folder org following Python best practices?
2. Set up a FastAPI backend with: CORS middleware for React frontend (localhost:5173), router structure for /api/files, /api/analysis, /api/graphs, /api/stats, WebSocket endpoint for realtime progress updates, global exception handler, proper project structure with routes in seperate files. Include imports and basic health check endpoint.
3. Create a React + Vite + TypeScript project structure with: Material-UI integration, React Router for multi-page nav (Dashboard, Demo, Settings), dark gradient theme styling, API service layer w/ axios, component structure for spectral analysis UI. Generate package.json, tsconfig.json, vite.config.ts, and App.tsx with routing.
4. I need a configuration mgmt system in Python using dataclasses for: API settings (OpenRouter API key, model selection), cache settings (TTL, limits, Redis config), UI settings (theme, window state), file paths and defaults. Include JSON serialization/deserialization and a ConfigManager class with get/set methods. Store config in users home directory.
5. Build a robust CSV parser for spectroscopy data that handles: Automatic delimiter detection (comma, tab, semicolon), encoding detection (UTF-8, Latin-1,

etc.), header detection, European vs US number formats (comma vs period decimals), comment line detection (#, //, ;, etc.), multi-column formats, metadata row identification. Use pandas for data handling, chardet for encoding. Return structured result with success flag, data, format info, and issues list.

6. Organize this spectral analyzer project into proper Python package structure: spectral_analyzer/ with core/ (CSV parsing, AI normalization, graph gen), ui/ w/ components/ and dialogs/, utils/ (helpers, caching, security, logging), config/, resources/, and tests/. Create all init.py files w proper imports for clean package access.
7. Create an API client for OpenRouter with: Exponential backoff retry (3 attempts), rate limiting (5 req/sec), request/response logging, error categorization (network, auth, rate limit, API error), timeout handling (30s default), cost tracking integration hooks. Use httpx for async HTTP. Include APIResponse dataclass for structured results.
8. Implement secure API key storage using: System keyring for production (keyring library), encrypted file fallback if keyring unavailable, environment variable support for development, key validation before storage, clear error msgs for missing keys. Create SecureKeyManager class with get_key, set_key, delete_key methods.
9. Build a spectroscopy data validator that checks: Wavenumber range (400-4000 cm^{-1}), wavenumber ordering (should be descending), absorbance range (0-5 typical), missing values and data gaps, duplicate wavenumbers, data spikes and anomalies (statistical outliers), minimum data points (atleast 100 for valid spectrum). Return validation results with severity levels (error, warning, info) and specific issue descriptions.

10. Create a professional graph generator for spectral data: Baseline + Sample overlay comparison graphs, high-quality export (300 DPI PNG/PDF), IR spectroscopy conventions (inverted x-axis, 4000→400), professional styling (clean grid, proper labels, legend), batch processing support (multiple samples vs one baseline), memory-efficient processing (close figures after saving), safe filename generation (no conflicts). Use matplotlib with Figure/FigureCanvas for PyQt integration.
11. Implement a 3-tier caching system: Tier 1 - Memory (LRU cache with 1000 entry limit), Tier 2 - File (compressed JSON with metadata), Tier 3 - Redis (optional distributed cache). Features: TTL-based expiration (24h default), background cleanup thread, compression (gzip/lz4) for files >10KB, SQLite metadata database, cache stats (hit rate, sizes, timing), file structure hashing for cache keys. Use threading.Lock for thread safety.
12. Build a cost tracking system for AI API usage: Track tokens, cost per call, total spending, budget alerts (warn at 80%, alert at 100%), cost breakdown by model and operation type, cache hit savings calc, daily/weekly spending reports, export to CSV for reporting, SQLite storage for history. Include CostTracker class w alert threshold config.
13. Create a secure file manager for handling CSV uploads: Path traversal prevention (validate paths stay in workspace), file size limits (50MB max), allowed extensions whitelist (.csv, .txt, .dat), temp file cleanup on exit, unique filename generation for conflicts, MIME type validation, safe file ops with error handling. Include FileManager class w validate_path, save_upload, cleanup methods.
14. Implement batch processing for multiple CSV files: Parallel processing w ThreadPoolExecutor (4 workers), progress callback system (message, percentage), error collection (continue on individual failures), result aggregation

(successful, failed, warnings), memory mgmt (process and release), timing metrics (total time, avg per file), graceful shutdown handling. Include BatchProcessor class with process_batch async method.

15. Create the infrastructure for AI-powered CSV normalization (I'll implement the core algorithm myself): NormalizationPlan dataclass (column mappings, transformations, confidence), confidence levels enum (HIGH >90%, MEDIUM 70-90%, LOW <70%), request/response structures for AI API, usage stats tracking, fallback plan generation (for when AI fails), plan serialization for caching. Don't implement the AI prompt or analysis logic yet - just the data structures and scaffolding.
16. Implement transformation functions for spectroscopy data normalization: Sort by wavenumber (ascending/descending), convert transmittance ↔ absorbance ($A = -\log_{10}(T/100)$), remove duplicate wavenumbers, interpolate missing values (linear), remove/clip negative values, normalize intensity (0-1 scaling), scale by custom factor, skip header rows, remove specified columns, reverse data order, outlier removal (IQR method). Each transformation should handle errors gracefully and log actions.
17. Create OpenRouter-specific API client: Model registry (Claude, GPT-4, etc. with pricing), streaming response support, token counting and cost estimation, model recommendation by task type (normalization, interpretation), response parsing and validation, error handling for OpenRouter-specific errors, test connection method. Include model pricing data and automatic cost calc.
18. Build comprehensive error handling system: Custom exception hierarchy (ValidationError, APIError, CacheError, etc.), error context preservation (stack traces, request data), user-friendly error messages, error categorization (recoverable vs fatal), logging integration, error reporting helpers, retry decision

logic. Create decorators for automatic error handling and logging.

19. Set up structured logging system: Rotating file logs (10MB max, 5 backups), console output w color coding, log levels by module (DEBUG for dev, INFO for prod), performance timing decorators, structured log formatting (timestamp, level, module, message), separate logs for errors, API calls, cache hits, log sanitization (remove API keys). Configure using Python logging w custom formatters.
20. Create CSV preview generator for AI analysis: Limit to 50 rows for token efficiency, truncate at 8000 chars to avoid context limits, include column info and sample values, preserve data types and formats, add truncation indicator if needed, generate structured summary (rows, cols, dtypes). Return formatted string suitable for LLM prompt inclusion.
21. Create FastAPI file upload endpoint: `@router.post('/upload')` async def `upload_file(file: UploadFile)`. Accept CSV files, validate file size and type, save to temp directory, return file ID and metadata, handle multiple files, progress tracking for large files, cleanup old uploads (24h retention). Include proper error responses (400, 413, 415, 500).
22. Build graph generation endpoint: `@router.post('/graphs/generate')` async def `generate_graph(baseline_id, sample_ids, format)`. Accept file IDs for baseline and samples, generate comparison graphs, support PNG/PDF/SVG formats, return download URLs or file paths, progress updates via WebSocket, batch graph generation, error handling for invalid files. Use background tasks for long ops.
23. Create analysis API endpoints: `/api/analysis/normalize` (AI normalization), `/api/analysis/validate` (data validation), `/api/analysis/color` (grease color analysis), `/api/analysis/interpretation` (AI interpretation). Each should: accept file ID or direct data, return structured results, include confidence scores, handle errors

gracefully, support async processing, cache results. Use Pydantic models for request/response validation.

24. build monitoring endpoints: `/api/stats/cache` (cache performance metrics), `/api/stats/costs` (API cost tracking), `/api/stats/usage` (system usage statistics), `/api/stats/health` (health check w dependencies). Return: hit rates/sizes/timings, cost breakdowns/alerts, memory/disk usage, service status. Format as JSON w proper status codes.
25. Configure FastAPI middleware: CORS for localhost:5173, localhost:3000, request logging middleware, response compression, request ID tracking, timing middleware (X-Response-Time header), error handling middleware. Apply in correct order and configure for development.
26. Implement WebSocket connection manager for progress updates: Session-based connections, progress message format: `{type, message, progress}`, connection pooling per session, graceful disconnect handling, dead connection cleanup, broadcast to session subscribers. Create `ConnectionManager` class w `connect`, `disconnect`, `send_progress` methods.
27. Create demo data generation endpoint for testing: Generate realistic spectral data, multiple format variations (clean, noisy, European format, etc.), return as downloadable CSV, include metadata and documentation, various complexity levels (simple, medium, complex). Use numpy for realistic spectral curves w noise.
28. Create main Dashboard page component: File upload area w drag-and-drop, baseline file selector (single), sample files selector (multiple), analysis controls (normalize, validate, generate graphs), results display area, progress indicators, error display. Use Material-UI Card, Button, CircularProgress. Connect to API

service layer.

29. Build drag-and-drop file upload component: Visual drop zone w hover effects, file preview cards, upload progress bars, file validation (CSV only, size limits), remove file functionality, multiple file support, visual feedback for drag states. Use react-dropzone or implement with native drag events. Style w MUI.
30. Create GraphCard component: Thumbnail preview of graph, sample name and metadata, download button (PNG/PDF), full-size modal on click, loading state, error state display, responsive layout. Props: graphUrl, sampleName, metadata, onDownload. Use MUI Card and Dialog.
31. Build GreaseColorPanel component to display color analysis: Color swatch (actual computed color), RGB values display, color description text, spectral feature breakdown (C-H stretch, carbonyl, etc.), oxidation level indicator, confidence bars, analysis notes. Props: colorData {rgb, hex, description, analysis}. Use MUI Paper and Typography.
32. Create InterpretationReportView component: Markdown rendering for AI text, structured sections (summary, findings, recommendations), confidence indicators, export to PDF button, print-friendly styling, expandable sections. Use react-markdown for content rendering. Style for readability.
33. Create TypeScript API service module: `// services/api.ts export const api = {uploadFile, normalizeData, generateGraph, analyzeColor, getInterpretation, getStats}`. Axios-based HTTP client, TypeScript interfaces for responses, error handling w user-friendly messages, request/response interceptors, loading state management, abort controller for cancellation. Configure base URL from environment.

34. Build Settings page component: API key mgmt (masked input), model selection dropdown, cache configuration (TTL, limits), theme selection, export settings, cost limits config, save/reset buttons. Use MUI TextField, Select, Switch. Store in localStorage temporarily.
35. Implement toast notification system: Success, error, warning, info types, auto-dismiss after 5s, stacking multiple toasts, action buttons (undo, dismiss), position top-right, slide-in animation. Use MUI Snackbar or create custom w/ framer-motion.
36. Create full-screen graph modal: Large graph display, zoom controls, download options (PNG, PDF, SVG), metadata sidebar, close button, keyboard navigation (ESC to close), swipe between multiple graphs. Use MUI Dialog w/ fullScreen prop. Add image zoom functionality.
37. Create modern PyQt6 main window: Menu bar (File, Edit, Process, View, Help), toolbar w/ icon buttons, splitter layout (input panel | preview panel), status bar w/ progress, dark theme support, card-based content areas, drag-and-drop zones. Use QMainWindow, QSplitter, QMenuBar. Apply modern styling.
38. Build PyQt6 drag-and-drop file zone: Visual drop area w/ border, hover state styling, file validation on drop, selected files list, remove file buttons, browse button fallback, file type filtering (CSV), size limit handling. Inherit QWidget, implement dragEnterEvent, dropEvent. Emit files_dropped signal.
39. Create spectral data preview widget: Embedded matplotlib graph (FigureCanvas), data table view (QTableView), tab switching between graph and table, zoom/pan controls, export actions, statistics panel, refresh button. Use QWidget w/ QVBoxLayout, integrate matplotlib FigureCanvasQTAgg.

40. Generate Qt Style Sheet (QSS) for dark theme: Professional color palette (dark grays, accent blues), button styling (hover, pressed states), input fields (borders, focus), menu and toolbar styling, card containers (elevation effect), scrollbar customization, consistent spacing and borders. Save as .qss file for application-wide loading.
41. Implement desktop toast notifications: Slide-in from top-right, auto-dismiss timer (5s), success, error, warning, info styles, icon indicators, close button, stacking multiple toasts, fade-out animation. Use QPropertyAnimation for smooth transitions. Create ToastNotification class.
42. Build enhanced QStatusBar: Status message area, progress bar (show/hide), operation indicator, cache stats display, API status indicator, cost display widget, icon indicators. Extend QStatusBar w custom widgets. Update from signals.
43. Create AI settings config dialog: API key input (masked), model selection combo box, temperature slider (0-1), max tokens input, cost limit settings, test connection button, save/cancel buttons, validation before save. Use QDialog w form layout. Emit settings_changed signal on save.
44. Build file preview dialog: Large graph display, data statistics, column information, normalization preview, accept/reject buttons, copy to clipboard, export options. Use QDialog w embedded matplotlib. Show before/after comparison.
45. Create cost monitoring display widget: Current session cost, total cost counter, budget progress bar, alert indicators, cost breakdown button (opens detailed view), auto-update every 5s. Use QWidget w QLabel, QProgressBar. Connect to cost tracker signals.
46. Set up pytest-based integration test suite: Test fixtures for sample data, mock API responses, database test setup/teardown, async test support

(pytest-asyncio), coverage reporting (pytest-cov), test data generators, assertion helpers. Create conftest.py w shared fixtures.

47. Write integration tests for CSV parser: Standard format parsing, European decimal format, tab-delimited files, missing headers, comment lines, encoding variations, malformed data handling, empty file handling. Test multiple real-world scenarios. Use parametrize for multiple cases.
48. Create realistic test data generator: Generate synthetic spectral data (numpy), various CSV formats (standard, European, problematic), different header styles, noise and outliers, missing values, edge cases (empty, single row, huge files). Save as CSV files in tests/test_data/ directory.
49. Write tests for API client: Successful requests, retry logic (w mock failures), rate limiting behavior, timeout handling, error categorization, response parsing. Use pytest-mock for API mocking. Test both sync and async methods.
50. Test multi-tier cache: Cache hit/miss scenarios, TTL expiration, LRU eviction, compression/decompression, thread safety (concurrent access), cache statistics accuracy, cleanup processes. Use temp directories for file cache tests.
51. Create comprehensive E2E test: Upload CSV file, parse and validate, AI normalize (w mock), generate graph, analyze color, get interpretation, export results. Test complete workflow w real file I/O. Clean up after test.
52. Create demo.py script showcasing features: Load sample spectral data, run normalization, generate comparison graphs, show color analysis, display cost stats, print cache performance, beautiful terminal output (colors, tables). Make it runnable w/ python demo.py. Use rich library for formatting.

53. Test error handling accross components: Invalid file formats, API failures (timeout, auth, rate limit), cache failures, disk full scenarios, network errors, corrupted data. Verify error messages, logging, and recovery mechanisms.
54. Create performance benchmark script: Parse performance (various file sizes), cache lookup speed, graph generation time, batch processing throughput, memory usage tracking, export results to JSON. Use timeit and memory_profiler. Generate report comparing optimized vs baseline.
55. Debug: CSV parser failing on European decimals (1,234 instead of 1.234). Current code attempts str.replace but doesnt handle thousands separator correctly. Need to: detect decimal seperator (. vs ,), handle thousands separator properly, convert to float correctly, preserve precision. Fix in `_convert_european_numbers` method.
56. Fix: Batch graph generation consuming excessive memory. Issue: matplotlib figures not being closed after saving. Need to: explicitly close figures w `plt.close(fig)`, clear axes between graphs, monitor memory in batch loop, add garbage collection hints. Update `generate_batch_graphs` method.
57. Debug: WebSocket disconnecting unexpectedly during long operations. Add: ping/pong keepalive mechanism, reconnection logic on client, dead connection detection on server, connection state mgmt, timeout configuration. Update WebSocket endpoint and ConnectionManager.
58. Add PDF export functionality: Convert interpretation report to PDF, include graphs as embedded images, proper formatting and page breaks, header/footer w metadata, table of contents. Use reportlab or weasyprint. Create `export_to_pdf` function returning file path.

59. Implement batch download as ZIP: Collect all graphs for a session, include analysis reports (JSON/CSV), add README w metadata, compress into ZIP archive, return download link, auto-cleanup after 24h. Create `archive_results` function using `zipfile`.
60. Make error messages more user-friendly: Technical errors → Plain English, include suggested actions, add error codes for support, link to documentation where relevant, show context (what was being attempted). Update error handling to use new message templates.
61. Add proper loading indicators: Skeleton loaders for data tables, spinner for API calls, progress bars for file upload, shimmer effect for graph loading, disable buttons during processing, optimistic UI updates. Use MUI `Skeleton` and `CircularProgress`. Add to all async ops.
62. Add accessibility features: ARIA labels for interactive elements, keyboard navigation support, focus indicators, screen reader alt text for graphs, color contrast compliance (WCAG AA), skip to content links. Audit w axe-core and fix violations.
63. Add docstrings to all public methods: Google-style docstrings, type hints in docstrings, example usage where helpful, link to related methods, note any side effects. Focus on public API methods and complex algos.
64. Create comprehensive README.md: Project overview and purpose, features list w emojis, installation instructions (desktop + web), quick start guide, configuration guide, API documentation links, screenshots (placeholders), architecture overview, contributing guidelines, license information. Make it visually appealing and easy to scan.

