Stage 1

Description

The application aims to solve the challenge of efficiently locating specific images stored on a user's local system by enabling intelligent, AI-powered search capabilities. Traditional file-based search methods are limited to filenames and basic metadata, making it difficult to find images based on content or context. This desktop-based solution will integrate Vision-Language Models (VLMs) and Large Language Models (LLMs) to allow text-based, image-based, and context-aware searches. Users can refine searches with filters like date range, location, resolution, objects, or scene types for more precise results. The system will support advanced features such as auto-tagging, metadata extraction, duplicate detection, and multimodal search while ensuring fast, efficient performance through indexing and caching. The application will prioritize privacy by processing all data locally and providing encryption options. Designed for usability, it will feature an intuitive GUI, cross-platform support, and extensibility for future enhancements, making it a comprehensive and intelligent local image search solution.

Goals

Dual-Mode Search Functionality:

- Text-Based Search: Implement a robust search engine that allows users to find images using descriptive text queries. This involves analyzing image metadata and content to match user inputs effectively.
- Image-Based Search: Enable users to input a reference image to find similar images within their local storage. This requires developing or integrating algorithms capable of assessing visual similarities.

Integration of Advanced AI Models:

- Vision-Language Models (VLMs): Utilize VLMs to understand and correlate visual content with textual descriptions, enhancing the accuracy of search results. Models like BLIP-2, which employ frozen image encoders and LLMs, can be considered for their efficiency and performance. HUGGINGFACE.COM
- Large Language Models (LLMs): Incorporate LLMs to interpret complex user queries, allowing for more natural and conversational search inputs.

User Interface and Experience:

- Intuitive Design: Develop a clean, responsive, and accessible graphical user interface (GUI) that caters to users with varying levels of technical expertise, ensuring ease of use in performing searches and viewing results.
- Real-Time Feedback: Provide immediate visual feedback during search operations, such as displaying thumbnails of search results as the user types.
- Search Filters and Sorting: Provide options to filter and sort search results based on various criteria such as date, file size, resolution, and more.

Performance Optimization:

- Efficient Indexing: Implement indexing mechanisms to catalog images and their attributes, reducing search times and improving overall performance.
- Resource Management: Optimize the application to function smoothly on a range of hardware configurations, ensuring it does not excessively consume system resources.

Privacy and Security:

- Local Processing: Ensure all image data and search operations are processed locally, eliminating the need for internet connectivity and safeguarding user privacy.
- Data Encryption: Incorporate encryption protocols to protect sensitive image data and user information from unauthorized access.

Extensibility and Integration:

- Plugin Support: Allow for third-party plugins to extend the application's functionality, enabling features like support for additional image formats or integration with other software.
- API Access: Provide an API to facilitate integration with other applications, allowing users to incorporate the image search functionality into their workflows seamlessly.

Continuous Learning and Adaptation:

- User Feedback Loop: Incorporate mechanisms for users to provide feedback on search results, enabling the application to learn and improve its accuracy over time.
- Model Updates: Regularly update the integrated AI models to incorporate the latest advancements in VLM and LLM research, ensuring the application remains state-of-the-art.

Market research

The market for AI-powered local image search applications is expanding, driven by the increasing need for efficient image retrieval solutions. Existing desktop search tools like Lookeen and Copernic Desktop Search primarily focus on text-based file retrieval and lack advanced image search capabilities. While some applications offer basic image search functions, they often do not fully leverage AI technologies for comprehensive search experiences. Competitors such as Quark have integrated AI recognition to enhance image search and organization, allowing users to search for photographs based on content rather than keywords or captions.
https://www.quark.com/about/blog/use-ai-recognition-in-local-image-libraries-to-easily-search-for-images
Additionally, open-source projects like LocalAI enable local AI inferencing, allowing users to run large language models and generate images without the need for a GPU.
https://itsfoss.com/local-ai-image-tools/ . Potential customers for an AI-powered local image search application include photographers, designers, content creators, and businesses managing extensive image libraries who require efficient tools to locate specific images based on content and context. The global image recognition market is projected to grow significantly, reaching an estimated value of US$ 146.10 billion by 2032, indicating a strong demand for

advanced image search solutions.
https://www.coherentmarketinsights.com/industry-reports/image-recognition-market
The market for AI-powered local image search applications is experiencing significant growth, driven by the increasing need for efficient and intelligent image retrieval solutions across various industries. Existing desktop search tools, such as Lookeen and Copernic Desktop Search, primarily focus on text-based file retrieval and often lack advanced image search capabilities. While some applications offer basic image search functions, they may not fully leverage the potential of advanced AI models for comprehensive search experiences. Several competitors have emerged, integrating AI technologies to enhance image search functionalities. For instance, Quark has developed features that utilize AI recognition to improve image search and organization, allowing users to locate photographs based on content rather than relying solely on keywords or captions. Open-source projects like LocalAI provide alternatives by enabling local AI inferencing on consumer-grade hardware, allowing users to run large language models (LLMs) and generate images without the need for a GPU. The global image recognition market is projected to grow significantly, with estimates suggesting it will reach approximately US$ 146.10 billion by 2032, growing at a compound annual growth rate (CAGR) of 15.5% from 2025 to 2032. This growth is driven by the increasing demand for AI-powered solutions across various sectors, including healthcare, retail, and security, where efficient image analysis and retrieval are critical. Potential customers for an AI-powered local image search application include photographers, designers, content creators, and businesses managing extensive image libraries. These users require efficient tools to locate specific images based on content and context, moving beyond traditional search methods that rely on filenames and basic metadata. The integration of Vision-Language Models (VLMs) and LLMs can significantly enhance the accuracy and efficiency of image retrieval, providing a more intuitive and powerful search experience.

Risks

Technical Risks:

- Scalability Issues: Handling large-scale image libraries efficiently without slowing down search performance.
- Model Latency & Optimization: Running VLMs and LLMs locally can be resource-intensive; optimizing for speed without sacrificing accuracy is a challenge.
- Edge Cases & Errors: AI models may misinterpret images, fail to recognize certain objects, or struggle with complex queries.
- Hardware Constraints: Some users may not have high-end hardware (GPU, high RAM), affecting the app's usability.
- Data Corruption & Loss: Improper indexing or database failures could lead to lost metadata or inaccurate search results.
- File System Access: Ensuring smooth scanning and searching across different file systems and external drives.

Financial & Resource Risks:

- Development Costs: High computational demands may require investing in optimized AI models or external libraries.

- Time to Market: AI-powered applications take longer to develop, test, and refine compared to traditional software.
- Competitor Advantage: Existing solutions might already dominate the market, making differentiation crucial.
- Monetization Challenges: Finding a sustainable revenue model (free vs. paid features, subscriptions, etc.).
- Ongoing Maintenance Costs: Updating AI models, fixing bugs, and optimizing performance requires long-term commitment.

Security & Privacy Risks:

- User Data Privacy: Ensuring that all processing remains local and user data is not exposed.
- Security Vulnerabilities: Preventing unauthorized access to image data or indexed search results.
- AI Bias & Ethical Risks: The model might generate biased or misleading metadata/tags, affecting search accuracy.
- Legal Compliance: Ensuring adherence to copyright laws when dealing with AI-generated metadata.

User Experience Risks:

- Complexity vs. Simplicity: Balancing advanced AI features with an intuitive user experience.
- Search Accuracy Perception: Users may have high expectations for accuracy, making misinterpretations frustrating.
- Learning Curve: Users unfamiliar with AI-powered search might struggle with advanced features.
- Customization Needs: Different users may have unique preferences for search filters, requiring flexible settings.

Integration & Extensibility Risks:

- Future-Proofing: Ensuring the architecture supports easy integration with new AI models.
- Compatibility with Cloud Storage: If users request cloud-based image search later, integration could be complex.
- API & Plugin Support: If extensibility is planned, designing a modular system from the start is necessary.

Stage 2

Functional

Image Indexing:

- Automatically index images from user-designated directories.
- Extract metadata (EXIF data, resolution, file type) for efficient search.
- Support for various image formats (JPEG, PNG, GIF, etc.).

Search Modes:
- Text-Based Search: Support natural language queries to search for images by description (e.g., "beach at sunset").
- Image-Based Search: Drag-and-drop or file selection for searching with a reference image. Reverse image search to find similar or identical images.
- Context-Based Search: Search based on scene, objects, or semantic meaning (e.g., "image of a car on the street"). AI-based interpretation for complex queries.
- Multimodal Search

Filtering Options:
- Date Range Filter: Search images within a specific timeframe.
- Location Filter: Filter by location metadata (GPS, EXIF data).
- File Type Filter: Filter based on image format (JPEG, PNG, etc.).
- Resolution Filter: Search by image resolution (HD, 4K).
- Orientation Filter: Filter by portrait, landscape, or square orientation.
- Color Filter: Filter by dominant colors or color palette.
- Object/Scene Filter: Search images containing specific objects (e.g., “trees,” “people”).
- Custom Tags: Users can create and filter by custom tags.
- Resolution filter (high, medium, low)
- Aspect ratio filter (portrait, landscape, square)
- Color-based search (dominant colors)
- Image size filter (small, medium, large)
- Blur/sharpness filter
- Object-based filtering (search images containing specific objects)
- Face detection filter (search for images containing faces)
- Scene detection (indoor, outdoor, nature, urban, etc.)

Metadata & Custom Tagging:
- Auto-extraction of metadata (EXIF, IPTC, XMP)
- User-defined tags and categories
- AI-generated captions/descriptions for images
- Image rating system for better organization
- Keyword-based tagging
- Auto-Tagging: Use AI models for automatic tagging based on content.
- Manual Tagging: Users can add custom tags to images.
- OCR (Optical Character Recognition): Extract text from images for searchability.

Graphical User Interface (GUI):
- Intuitive, user-friendly design supporting novice and advanced users.
- Multiple view options: grid, list, and detailed views.

- Dark mode/light mode UI options
- Multi-language support
- Search History & Recent Queries: Ability to view and reuse past searches for quicker retrieval.
- Drag-and-Drop Functionality: Users can drag an image to initiate a search for similar images.

Search Results Interaction:

- Allow users to rename, move, or delete images from results.
- Bookmark images as favorites for quick access.
- Batch search for multiple images at once
- Thumbnail preview in search results
- Search result ranking based on relevance

Interactivity & Exporting:

- Ability to edit and refine search queries interactively
- Export search results as lists or folders
- Sharing functionality (copy results to clipboard, send via email)
- Integration with cloud storage (optional)

Privacy & Security:

- Local Processing: All data is processed locally without external transmission.
- Data Encryption: Ensure metadata and image data are securely stored.
- Permission & Consent: User permission required to scan specific directories or folders.
- Secure File Management: Secure image deletion options, ensuring complete removal of sensitive files.
- Secure deletion option (permanent file removal)

Performance & Optimization:

- Indexing Efficiency: Optimized indexing for large image libraries.
- Caching: Implement caching for frequently accessed images to speed up results.
- Batch Processing: Allow users to bulk select and manage images for actions like renaming, tagging, or moving.
- Precomputed embeddings for efficient retrieval
- Duplicate image detection and removal
- Image grouping/clustering based on similarity
- Resource Management: Ensure smooth operation on both low-end and high-end systems.

Advanced AI Capabilities:

- AI-based OCR (text recognition within images)
- Handwritten text detection in images
- Emotion detection in images
- Scene change detection in sequential images

- AI-powered object tracking across multiple images
- Custom model training for personalized search improvements

Extensibility & Updates:
- Plugin Support: Modular system to allow for future enhancements or third-party integrations.
- AI Model Updates: Allow easy integration of new AI models for improved search accuracy.
- Customizable Settings: Users can configure indexing frequency, search filters, and UI preferences.
- API access for automation and external integration

Business Rules:
- Data Privacy: Local data processing ensures privacy; no data is shared externally. User data and images are encrypted for added security.
- Directory Access: Users must give explicit consent to scan directories or folders. Follow file system permissions to respect user privacy.
- Compliance: Ensure the application complies with relevant data protection regulations (e.g., GDPR).
- Monetization Model: Free version with essential features. Premium features like advanced search filters, plugin support, and AI enhancements available via subscription or one-time purchase.

Error Handling & Feedback:
- Error Logging: Implement robust error logging and monitoring to detect issues. Provide users with error reports for troubleshooting.
- User Feedback: Allow users to submit feedback or suggestions for improving search accuracy or features. Use feedback loops to refine search algorithms and improve user experience.

Non-functional

Performance:
- Search Speed: Search results should be delivered within 2-3 seconds for small datasets and no more than 10-15 seconds for large datasets (over 10,000 images).
- Low Latency for AI Models: AI models (VLMs, LLMs) should process queries with minimal delay, optimized for local resource use, with optional GPU support for faster inference.
- Indexing Efficiency: Image libraries must be indexed efficiently, allowing incremental updates without causing performance degradation. Support for indexing multiple directories simultaneously, handling new and modified images swiftly.
- Memory & CPU Usage: The application must be optimized to run on both low-end and high-end hardware, utilizing minimal CPU and memory resources while performing operations like indexing and searching. Background tasks (indexing, cache updates) should not impact foreground performance significantly.

Security:

- Data Encryption: Ensure all user data, including image metadata and tags, is encrypted using industry-standard encryption protocols (e.g., AES-256).
- Local Processing: All image processing and AI model inference should occur locally on the user's machine to ensure that sensitive data never leaves the device.
- Access Control: Implement strong authentication methods to restrict access to the application and its sensitive data (e.g., password protection or user account management).
- Secure File Management: Provide users with options for securely deleting images, ensuring no recoverable traces are left on the system.
- Backup & Restore: Secure backup and restore functionalities should be implemented, allowing users to recover indexed data and metadata securely in case of system failure.

Scalability:

- Handling Large Datasets: The system must efficiently handle growing image collections, potentially in the millions, without significant slowdowns. Indexing and search processes should scale horizontally if cloud options are added later or vertically to accommodate local machine resources (e.g., multi-core CPU/GPU support).
- Concurrent User Support (for future versions): If multi-user functionality is considered in the future, the application should support multiple users operating simultaneously, ensuring no performance degradation.
- Model Scalability: The AI models used should be scalable to allow for updates, new models, or more resource-heavy versions (such as models that process videos or high-resolution images).
- Distributed Search (for future versions): If cloud storage is added, support for distributed search capabilities should be included, allowing for faster and more scalable searches across multiple systems.

Accessibility:

- Cross-Platform Compatibility: The application must be compatible with Windows, macOS, and Linux to accommodate a broad user base.
- UI Accessibility: Implement features like keyboard shortcuts, screen reader support, high-contrast modes, and customizable font sizes to ensure that the application is usable for people with disabilities.
- Responsive Design: The user interface should adapt to different screen sizes and resolutions, ensuring a consistent experience across devices (e.g., laptops, desktops).
- Internationalization & Localization: Support multiple languages and regional preferences to ensure global accessibility.
- Error Messaging & Guidance: Provide clear and concise error messages with actionable solutions to help users resolve issues independently.

Compliance:

- Data Privacy Regulations: The application must comply with relevant data privacy laws and regulations, such as the General Data Protection Regulation (GDPR), California Consumer Privacy Act (CCPA), and Health Insurance Portability and Accountability Act (HIPAA) (if applicable).
- Content & Copyright Laws: Ensure that the application adheres to copyright laws, including guidelines for handling copyrighted images (e.g., proper attribution, permissions for use, etc.).
- Software Licensing: Ensure compliance with open-source or third-party software licenses used in the development of the application.
- Data Retention & Deletion: Comply with data retention policies, allowing users to delete indexed images or metadata upon request or after a specified period.
- Ethical AI Use: The AI models used should be trained and deployed responsibly, ensuring that the image recognition process is unbiased and ethical.

Stage 3

High-Level Architectural Planning
Architecture Style: For a desktop application with local processing, a modular monolithic approach simplifies deployment and maintenance while allowing clear separation of concerns. This design avoids the overhead of microservices or serverless architectures, which are better suited to distributed or cloud environments, while still enabling modularity for future enhancements.

Tech Stack Selection:
Frontend: Framework:Electron.js. UI Technologies:HTML, CSS, and optionally React.js for dynamic UI components Electron.js enables the use of web technologies to create cross-platform desktop applications, and React can improve state management and responsiveness.
Backend: Language:Python, Framework:FastAPI (or Flask)
Database: Primary Choice:SQLite, Alternate:PostgreSQL (if scalability requirements increase). SQLite is lightweight and well-suited for local indexing and metadata storage, ensuring quick setup and minimal resource consumption.

Integration Strategy:
Internal Integration: Use RESTful APIs to connect the Electron frontend with the Python backend. This creates a clear, decoupled interface between UI and processing logic, easing maintenance and future updates.
Third-Party and Future Integrations: Design with extensibility in mind, allowing optional integrations with cloud storage, advanced AI services, or plugin architectures as needed.

API Design
Structure:REST API using FastAPI
Versioning Strategy:Implement semantic versioning to manage future updates
Documentation:Utilize Swagger UI (OpenAPI) for comprehensive API documentation
Reasoning:REST is widely adopted and easy to implement with FastAPI, and Swagger ensures clarity and ease of use for developers interfacing with the API.

Authentication & User Management APIs:
POST /api/users/register – Register a new user
POST /api/users/login – Authenticate user credentials
GET /api/users/{user_id} – Retrieve user details
PUT /api/users/{user_id} – Update user information
DELETE /api/users/{user_id} – Remove a user account

Image Management APIs:
GET /api/images – List images with query parameters (filters, pagination)
GET /api/images/{id} – Retrieve detailed image information
POST /api/images – Add new image entries (if manual addition is supported)
PUT /api/images/{id} – Update image metadata or attributes
DELETE /api/images/{id} – Delete an image record

Tagging APIs:
GET /api/tags – List all tags
POST /api/tags – Create a new tag
PUT /api/tags/{id} – Update an existing tag
DELETE /api/tags/{id} – Remove a tag
POST /api/images/{id}/tags – Associate a tag with an image
DELETE /api/images/{id}/tags/{tag_id} – Disassociate a tag from an image

Search APIs:
GET /api/search – Text-based search with support for filtering (e.g., date range, location, file type)
POST /api/search/image – Image-based search using an uploaded reference image
POST /api/search/context – Context-based search via chat prompt or descriptive query

Search History APIs:
GET /api/history – Retrieve past search queries and their results
DELETE /api/history/{id} – Delete specific search history records

Indexing & Log APIs:
POST /api/index – Trigger manual indexing of designated directories
GET /api/index-log – Retrieve logs of indexing operations and system updates

Settings & Configuration APIs:

GET /api/settings – Fetch current application settings
PUT /api/settings – Update settings (indexing paths, performance options, UI preferences)
GET /api/version – Retrieve application version and update information

Export APIs:
POST /api/export – Export image metadata or search results (e.g., CSV, JSON)
GET /api/export/history – List and track previous export operations

Feedback & Reporting APIs:
POST /api/feedback – Submit user feedback, bug reports, or suggestions

AI Model Integration APIs:
GET /api/ai/status – Check the operational status of AI models
POST /api/ai/infer – Manually trigger inference for testing image/text processing

Utility & Health Check APIs:
GET /api/health – Health check for the API server and underlying services
GET /api/metrics – Retrieve performance metrics or usage statistics (if applicable)

Database Design
Data Models:Create robust models for images, metadata (EXIF, tags, captions), and user preferences
Indexing Strategies: Index frequently queried fields (e.g., date, tags, file type).
Backup/Disaster Recovery: Include local export and backup functionalities

Image:
id (Primary Key)
file_path (Unique)
file_name
file_type
file_size
resolution (width, height)
creation_date / modification_date
metadata (JSON: EXIF data, auto-tags, captions, OCR results)
indexing_timestamp

Tag:
id (Primary Key)
tag_name

ImageTag (Association Table):
image_id (Foreign Key to Image.id)
tag_id (Foreign Key to Tag.id)

SearchHistory:
id (Primary Key)
query_text
search_type (enum: text, image, context)
filters_applied (JSON)
timestamp
result_count

Settings:
id (Primary Key)
key
value (string or JSON)

IndexLog:
id (Primary Key)
operation (e.g., indexing, update, delete)
timestamp
details (JSON)

User (Optional):
id (Primary Key)
username
hashed_password
email
preferences (JSON)

ExportHistory (Optional):
id (Primary Key)
export_type (CSV, JSON, etc.)
export_date
file_path

Queries:

- Basic SELECT queries filtering images by file_name, type, date range, etc.
- JOIN queries between Image and Tag via ImageTag for tag-based filtering.
- Full-text search queries on captions, descriptions using SQLite FTS5 extension.
- Range queries on creation_date/modification_date.
- Aggregation queries for indexing logs and performance metrics.

Relations & Schema:
Many-to-Many: Between Image and Tag (via ImageTag association table)
One-to-Many: Image to SearchHistory. User to Settings (if applicable)
Schema:

- Tables: images, tags, image_tags, search_history, settings, index_log, user (optional), export_history (optional)
- JSON columns for flexible metadata storage
- Indexes on frequently queried fields (file_path, file_name, creation_date)

Libraries & Tools:
ORM:SQLAlchemy (or Peewee as a lightweight alternative)
Database Engine:SQLite (for local storage) with support for JSON and FTS5
Migration Tool:Alembic
Additional: Use transaction management for bulk operations. Consider in-memory caching (if needed) for improved query performance

UI/UX Design
Style Guides: Establish clear guidelines for color schemes, typography, and overall visual aesthetics. Consider Material Design principles for consistency and modern appeal
Component Libraries: Utilize existing libraries like Material UI (if using React) or build a custom reusable component library

Accessibility Standards
Compliance:Adhere to WCAG guidelines
Features:Ensure support for screen readers, keyboard navigation, high-contrast themes, and adjustable font sizes

Responsive Layouts: Ensure adaptive design to accommodate various screen sizes and resolutions, even though the primary focus is desktop. Although it's a desktop application, responsive design principles help maintain a flexible and modern interface that adapts to different devices and display settings.

Splash/Startup Screen: Loading animation and branding display

Home/Dashboard Page:
- Overview of recent searches and indexed images
- Quick access to common functions (e.g., new search, settings)

Search Page/Main Interface:
- Search Bar:For text-based queries
- Drag-and-Drop Area:For image-based search input
- Chat Prompt/AI Assistant Widget:For interactive queries and context-based search
- Quick Filters Toolbar:For common filters (date, file type, tags)

Result Page:
- Thumbnail Grid/List View:Toggle between grid and list display of results
- Detailed Preview Pane:Image metadata, captions, and OCR text
- Sorting Options:By relevance, date, resolution, etc.

- Pagination/Infinite Scroll:For navigating large result sets
- Context Menu:Right-click options (rename, delete, tag, move)

Filter Dialog:
- Date range picker
- Location filter (based on metadata)
- File type, resolution, and orientation selectors
- Dominant color picker
- Custom tag and object/scene filters

Export Dialog:
- Options to select export format (CSV, JSON, image batch)
- Destination path selection
- Export progress indicator (progress bar)

Image Details/Metadata Page:
- Large image preview
- Display of metadata (EXIF, auto-generated tags, captions, OCR results)
- Edit options for tags and descriptions
- Action buttons (delete, move, share)

Settings/Preferences Page:
- Indexing Settings:Directory selection, scheduling indexing
- Performance Options:Caching, resource allocation
- Privacy & Security Settings:Data encryption, directory permissions
- Theme & Style Options:Light/dark mode, color scheme customization
- Notification Preferences:Alerts for errors, updates, and indexing progress

User Feedback & Error Dialogs:
- Feedback form for bug reports and suggestions
- Modal dialogs for error messages with troubleshooting tips

About/Help Page: User manual, FAQ, and troubleshooting guides, Application version and credits
Loading/Progress Indicators: Spinners or progress bars for long-running tasks (indexing, AI inference)
Sidebar Navigation/Menu: Navigation links to Dashboard, Search, Settings, Help, etc.
Notification System: Toast messages for success, error, and information alerts

Responsive and Accessible UI Components:
- Consistent style guides (typography, color schemes)
- High-contrast modes, keyboard shortcuts, and screen reader support
- Reusable component library for uniform UI elements across pages

Optional Authentication Components:

- Login/Registration dialogs (if premium or multi-user features are integrated)
- Account management section for user details and security settings

Stage 4

Frontend Development

- Component-Based Architecture: Develop reusable, modular UI components (e.g., search bar, filter dialog, result grid) using Electron.js with HTML, CSS, and React.js.
- Integrate a library such as Material UI or Bootstrap for consistent styling and rapid prototyping.

Routing & Navigation: Implement client-side routing (using React Router) for smooth navigation between pages (e.g., Dashboard, Search, Settings, Help).
State Management: Utilize Redux or Context API for managing global state (search queries, user settings) and clearly delineate local vs. global state.
Performance Optimization: Configure bundlers like Webpack or Vite for code splitting, tree shaking, and asset optimization (images, fonts).
Accessibility & Internationalization:

- Ensure adherence to WCAG guidelines with semantic markup, ARIA roles, and keyboard navigation.
- Integrate i18next for multilingual support.

Backend Development
Modular Project Architecture: Build the backend using Python and FastAPI, structuring code into controllers, services, and repositories.
Business Logic & Data Validation: Implement dedicated service layers for image indexing, search algorithms, and AI inference, using DTOs for data validation and transformation.
Security Best Practices: Integrate secure authentication (JWT or OAuth2), sanitize inputs, and enforce role-based access if multi-user features are needed. Encrypt sensitive data using AES/TLS.
Performance & Scalability:

- Employ asynchronous processing for tasks like indexing and AI model inference.
- Integrate caching (in-memory cache, e.g., Redis, if necessary) to reduce database load.

Database Design & Management
Indexing & Query Optimization: Create indexes on frequently queried fields (file_path, tags, dates) and utilize SQLite FTS5 for full-text searches on captions and descriptions.
Migrations & Versioning: Use Alembic for managing database schema migrations and version control for schema changes.
API & Integration

Third-Party & Internal Integration: Plan for internal integration (Electron frontend with FastAPI backend via REST and Electron IPC) and potential future integrations with cloud services or external AI tools.
DevOps & CI/CD
Version Control & Collaboration: Utilize Git with a branching strategy (e.g., Gitflow) on GitHub/GitLab for source control and code reviews.
Automated Build & Testing Pipelines: Set up CI/CD pipelines (using GitHub Actions, Jenkins, or GitLab CI) to automate unit, integration, and end-to-end tests (using Jest, React Testing Library, Cypress).
Deployment & Packaging: Package the desktop application using Electron Builder and bundle Python components using PyInstaller.
Monitoring, Logging & Alerts: Integrate logging (ELK stack or similar) and performance monitoring (e.g., Datadog, Prometheus) with alert systems for critical issues.
Configuration & Secrets Management: Manage configuration via environment variables and secure secrets using tools like HashiCorp Vault.

Stage 5

Unit Testing:

- Frontend: Use Jest and React Testing Library to test UI components, state updates, and event handling.
- Backend: Use PyTest for testing Python backend services, including API endpoints, image indexing, and AI model integrations.

Integration Testing:

- API Testing: Use Postman for testing RESTful API endpoints (CRUD operations, search queries, authentication, etc.).
- UI-Backend Integration: Use Cypress to verify that frontend components correctly interact with backend APIs.

End-to-End (E2E) Testing:

- Utilize Selenium or Playwright to simulate complete user workflows (e.g., launching the app, performing a search, and displaying results).
- Consider Electron-specific testing frameworks like Spectron if needed.

Security Testing:

- Vulnerability Scanning: Use automated tools like OWASP ZAP to perform vulnerability scans.
- API Security Testing: Validate proper authentication, input sanitization, and secure error handling in APIs.

Load Testing: Use JMeter or k6 to simulate concurrent requests and assess API and search performance.

Performance Testing: Test the indexing process, AI model inference latency, and overall application responsiveness on different hardware setups.
User Acceptance Testing (UAT): Engage target users (e.g., photographers, designers) to validate usability, functionality, and overall user experience.
Automated Testing: Automate repetitive tests (unit, integration, and E2E) to support continuous integration and regression testing.
Manual Testing: Perform exploratory, usability, and accessibility tests manually to capture issues not easily detected by automated tests.
CI Pipeline: Integrate all testing (unit, integration, E2E) into CI/CD pipelines (using GitHub Actions, Jenkins, or GitLab CI) to run tests on every commit.
Reporting: Generate automated code coverage and test reports to monitor quality.

Stage 6

Packaging & Build:

- Use Electron Builder for cross-platform packaging of the desktop application.
- Bundle Python components with PyInstaller to create standalone executables.

Update Mechanism & Distribution:

- Implement an auto-update system using Electron autoUpdater to check for new releases.
- Host installers and update packages on reliable storage services like AWS S3 or GitHub Releases.

Hosting Infrastructure for Supporting Services:

- If a companion website or documentation portal is needed, host frontend assets on platforms like Vercel/Netlify or AWS S3 with CloudFront CDN for fast global delivery.
- Set up a custom domain with SSL certificates (using Cloudflare or AWS Certificate Manager).

Backup & Disaster Recovery:

- Establish multi-region backups for installer binaries and update packages using services like AWS S3 with cross-region replication.
- Implement versioning on storage to enable rollbacks if necessary.

Rollback & Blue-Green Deployment Strategies: Use blue-green deployment techniques for update servers to ensure a smooth transition between application versions and allow quick rollback if issues arise.
CI/CD Integration:

- Configure automated build pipelines using GitHub Actions, GitLab CI, or Jenkins to automate packaging, testing, and deployment processes.
- Integrate automated tests (unit, integration, E2E) in the CI pipeline to ensure quality before deployment.

Security Measures in Deployment:

- Secure hosting with proper SSL/TLS configuration and regularly update security certificates.
- Monitor update servers and distribution channels for vulnerabilities using automated vulnerability scanning tools.

Monitoring & Logging: Set up monitoring for the update servers and application usage using tools like AWS CloudWatch or Datadog.