Bilkent University
Department of Computer Engineering

Senior Design Project
T2504
Pathogenius

Final Report

Nazlı Apaydın 22202104
Ege Ateş 22201914
Yiğit Ali Doğan 22202329
Yunus Günay 22203758
Ata Uzay Kuzey 22203050

Supervisor: Can Alkan
Course Instructors: Mert Bıçakçı, İlker Burak Kurt

04.05.2026

This report is submitted to the Department of Computer Engineering of Bilkent University in partial fulfilment of the requirements of the Senior Design Project course CS491/2.

Contents

1 Introduction

1.1 Purpose of the System

Recent advances in sequencing technologies have made it possible to analyze genetic material directly from clinical and environmental samples. The software ecosystems required to interpret this data, however, remain complex, computationally demanding, and largely dependent on high-performance servers or continuous cloud connectivity. In field hospitals, emergency response situations, and resource-limited clinics, which are precisely the settings where rapid pathogen identification is most valuable, these requirements become a barrier rather than an enabler.

Pathogenius addresses this gap with a desktop metagenomic analysis platform that runs on consumer-grade laptops and turns raw long-read FASTQ data into clear, species-level pathogen identifications. The analysis pipeline is orchestrated by Snakemake and built around the CLARK family of discriminative k-mer-based classifiers. By default, classification runs locally on the host machine and does not require an internet connection. When a CUDA compatible NVIDIA device is reachable over SSH, either a separate edge device such as a Jetson Nano or the host itself if the host has a CUDA GPU, the same analysis can instead be redirected to the GPU accelerated version of CLARK to shorten classification time on larger inputs. Both execution paths produce the same result format, so the rest of the system is unaffected by the choice.

The system is fronted by an Electron.js desktop application that hides command-line operations and shows real-time feedback on workflow stage and resource utilization. Once classification finishes, results are passed to a local AI interpretation layer built on a quantized medical language model running entirely on the host machine. This layer produces a clinical-style summary covering the main findings, antimicrobial resistance concerns, and suggested clinical actions, and streams the text into the results view as it is generated. No sample data, classification output, or interpretation is sent to any external service during analysis or summarization.

To support immediate use in the field, Pathogenius offers a Guest Mode that lets a user start an analysis without registering or logging in, with all data kept locally on the device. Users who do choose to register gain a persistent local account with email-based credential recovery and access to administrative features such as user management, but registration is not required to perform an analysis. The result is a system that behaves as a fully local diagnostic-support tool by default, with optional accounts available for teams that want them.

The completed system is positioned as a discovery-oriented decision support tool rather than a definitive diagnostic instrument. Its target users are clinicians, field workers, and researchers who need actionable species-level evidence from sequencing data without configuring a full bioinformatics environment.

1.2 Design Goals

The design of Pathogenius was guided by five quality goals derived from the project's non-functional requirements and the operational realities of its target environments. The categories below align with the ISO/IEC 25010 quality model used throughout the analysis and design phases.

1.2.1 Usability

Pathogenius is operable end to end through a graphical interface. No command-line interaction is required at any stage, and analyses are launched from a guided New Analysis screen that handles file selection and parameter configuration. Long-running analyses provide continuous visual feedback through progress indicators, workflow stage markers, and live system telemetry. Results are presented through interactive visualizations together with risk-color-coded pathogen detail cards, supplemented by an AI-generated summary that streams a clinical-style interpretation as it is produced. Navigation between the major functional areas is structured so that field personnel can complete a full analysis after minimal training.

1.2.2 Reliability

The full analysis path, from FASTQ ingestion through classification and AI summarization, runs on the local machine and does not require an internet connection. Required tools, the language model, and the reference database are packaged with the deployment so that the system can operate autonomously once installed. The pipeline is deterministic in the sense that identical inputs and reference databases produce identical classification output, and raw FASTQ files are treated as read-only throughout. Online features such as account creation and cross-device history sync rely on connectivity when a user opts into them, but the core analysis workflow remains fully usable in Guest Mode without any of these services.

1.2.3 Performance

The local pipeline is designed to run on mid-range consumer laptops, and large inputs can be split into smaller batches and merged at the abundance stage to keep memory usage manageable. When the user has access to a compatible GPU device, the same analysis can be routed to the GPU-accelerated CLARK variant for higher throughput. The system reports detected hardware on the dashboard so that users can make an informed decision about which execution path to use. The AI summarization layer is configured with bounded context and output sizes so that interpretation completes within a predictable time budget on a laptop CPU.

1.2.4 Maintainability

Pathogenius is structured as four modular layers covering the user interface, backend orchestration, the analysis workflow, and AI interpretation. The workflow is implemented as discrete Snakemake rules so that individual stages can be modified, replaced, or extended without affecting the rest of the system. The frontend uses standard Electron process separation between the main process, a sandboxed preload, and the renderer, which keeps platform integration concerns isolated from the UI code. The codebase is version-controlled, follows consistent coding conventions, and produces persistent session data that supports debugging and future development.

1.2.5 Security

Sequencing data is treated as sensitive throughout its lifecycle. Sample data, classification output, and AI summaries remain on the host machine during analysis. For users who choose to register, analysis

results stored on disk are encrypted at rest using authenticated symmetric encryption, with a per-user encryption key derived from the user's password through a salted, iterated key-derivation function. The key is held only in memory for the duration of the session and is cleared on logout. Credentials are handled through a managed authentication service that requires email verification before first login, and authentication tokens are persisted through the operating system keychain rather than plain configuration files. Access to administrative operations is gated behind role-based access control, enforced both inside the application and at the backing database's security rules, so that ordinary user accounts cannot reach user management functions. The desktop application itself follows Electron's recommended hardening posture, with the renderer process sandboxed and isolated from Node.js APIs and all communication between renderer and main process passing through a controlled bridge. Inputs that flow into file paths or workflow commands are validated against expected formats before use.

1.3 Definitions, Acronyms, and Abbreviations

- **AI**: Artificial Intelligence. In Pathogenius, the local language model used to convert structured classification results into a readable clinical summary.
- **AMR**: Antimicrobial Resistance. Resistance of a microorganism to drugs that would normally inhibit or kill it. AMR information is one of the categories surfaced in the AI summary.
- **CLARK**: A family of discriminative k-mer-based taxonomic classifiers used to assign sequencing reads to organisms from a reference database.
- **CUDA**: Compute Unified Device Architecture. NVIDIA's parallel computing platform, used by the GPU-accelerated variant of CLARK.
- **Electron.js**: A framework for building cross-platform desktop applications using JavaScript, HTML, and CSS, used to implement the Pathogenius frontend.
- **FASTA**: A text format for nucleotide or protein sequences, used to store reference genomes.
- **FASTQ**: A text format that stores nucleotide sequences together with per-base quality scores. The primary input format consumed by Pathogenius.
- **Guest Mode**: An offline access mode in which a user can run analyses without registering or signing in, with all data kept locally on the device.
- **GPU**: Graphics Processing Unit. A hardware accelerator for parallel computation.
- **GUI**: Graphical User Interface.
- **k-mer**: A substring of length k extracted from a DNA sequence and used as the unit of comparison in classifiers such as CLARK.
- **Metagenomics**: The study of genetic material recovered directly from environmental or clinical samples without isolating individual organisms.
- **NCBI**: National Center for Biotechnology Information. Source of the curated genomic data used to build local reference databases.
- **PCR**: Polymerase Chain Reaction. A target-specific DNA amplification technique used in conventional pathogen detection kits.
- **RBAC**: Role-Based Access Control. The model used in Pathogenius to differentiate regular users from administrators.
- **Snakemake**: The workflow management system used by Pathogenius to coordinate the stages of the analysis pipeline.
- **Taxonomic Classification**: The assignment of sequencing reads to known organisms based on similarity to a reference database.

1.4 Overview

This document is the final report of the Pathogenius project. It consolidates and refines the work documented in the Project Specification Document, the Analysis and Requirement Report, and the Detailed Design Report, and reports the outcome of the development, integration, and testing phases that followed.

Section 2 presents the final functional and non-functional requirements. Section 3 describes the final architecture and design, including the four-layer decomposition, hardware and software mapping, persistent data management, and access control. Section 4 covers the development and implementation of each subsystem along with the relevant API definitions. Section 5 reports the executed test cases and their outcomes. Section 6 describes the post-deployment maintenance plan. Section 7 covers the remaining project elements: engineering factors, ethics and professional responsibilities, teamwork, the extent to which the original project objectives were met, and the new knowledge acquired and applied during the project. Section 8 concludes with a summary and proposed future work, followed by the glossary in Section 9 and references in Section 10. A user manual accompanies this report as a separate deliverable, alongside the source code and the executable.

2 Requirements Details

This section presents the final functional and non-functional requirements of Pathogenius. Functional requirements describe the externally observable behavior of the platform, organized by feature area. Non-functional requirements describe the quality attributes the system must satisfy and follow the categories used in the Section 1 design goals (consistent with ISO/IEC 25010). A short Pseudo Requirements subsection at the end lists the binding technology choices. Each functional requirement is identified by a code of the form FR-XX-NNN, and each non-functional requirement by a code of the form NFR-XX-NNN, both for traceability to subsequent sections.

2.1 Functional Requirements

2.1.1 User Registration and Authentication

Authentication is optional. Pathogenius supports both registered accounts (with a persistent local workspace and access to optional cloud features) and an immediate offline Guest Mode that requires no credentials.

- **FR-UA-001**: The system shall allow new users to register an account with email, password, display name, and an optional institution field.
- **FR-UA-002**: The system shall require email verification before the user is permitted to log in.
- **FR-UA-003**: The system shall provide a "Resend verification email" action both during registration and after a failed login due to an unverified address.
- **FR-UA-004**: The system shall authenticate registered users through a username-and-password login interface.

- **FR-UA-005**: The system shall provide a "Forgot Password" action that sends a password-reset email to the address associated with an account, returning a generic success message regardless of whether the address is registered.
- **FR-UA-006**: The system shall allow an authenticated user to change their password from the Settings page, requiring the current password, a minimum length of 12 characters, a confirmation field, and rejection of a new password identical to the current one.
- **FR-UA-007**: The system shall provide a Guest Mode that grants immediate offline access without registration or login. Guest Mode shall provide access to the analysis pipeline, the Results page for analyses created in the same session, the reference Database management screen, and non-account-related Settings (AI summary toggle, analysis thresholds, notifications, appearance, settings export/import/reset, and system information). Guest Mode shall not expose account-security operations, cloud backup, or administrative functions.
- **FR-UA-008**: The system shall persist authenticated sessions across application restarts using the operating system keychain.
- **FR-UA-009**: The system shall log the user out on demand and clear in-memory session and key material on logout.
- **FR-UA-010**: The system shall prevent suspended accounts from logging in and shall display a clear message explaining the suspension state.
- **FR-UA-011**: The system shall maintain isolated per-user data so that one user cannot access another user's analyses or settings.
- **FR-UA-012**: Analysis results created in Guest Mode shall be stored locally in unencrypted form and shall be visible only within the active Guest Mode session. All Guest Mode data shall be cleared when the session ends.

2.1.2 Administration and Access Control

Pathogenius defines two roles, regular user and administrator, with role-based access enforced at both the application layer and the backing service.

- **FR-AD-001**: The system shall support exactly two user roles, `user` and `admin`, with the role stored on the authentication backend.
- **FR-AD-002**: The system shall enforce administrative privileges at both the application layer and the backing-database security rules so that role changes take effect immediately.
- **FR-AD-003**: The system shall provide an Administration panel (visible only to administrators) that lists all registered users with a search field.
- **FR-AD-004**: The system shall allow an administrator to suspend any user account, preventing login until reactivated.
- **FR-AD-005**: The system shall allow an administrator to reactivate a previously suspended account.
- **FR-AD-006**: The system shall allow an administrator to grant or revoke the administrator role on any account.
- **FR-AD-007**: The system shall allow an administrator to trigger a password-reset email for any user.
- **FR-AD-008**: The system shall allow an administrator to delete a user account.

- **FR-AD-009**: The system shall hide all administrative controls from non-administrator users in the user interface.

2.1.3 Analysis and Workflow Management

This subsection covers the lifecycle of an analysis as observed by the user, from creation through monitoring, control, and removal.

- **FR-AW-001**: The system shall allow the user to start a new analysis from the New Analysis screen.
- **FR-AW-002**: The system shall allow the user to configure per-analysis parameters before starting, including the sample name, the sample type, the reference database, and the classification engine (CPU or GPU).
- **FR-AW-003**: The system shall display the list of currently running analyses on both the Dashboard and the Results page.
- **FR-AW-004**: The system shall display the analysis history, listing all completed, failed, and cancelled analyses for the current user.
- **FR-AW-005**: The system shall allow the user to open a detailed result view for any completed analysis.
- **FR-AW-006**: The system shall allow the user to cancel a running analysis, terminating the underlying process tree.
- **FR-AW-007**: The system shall allow the user to delete an analysis from the history, removing the associated files and metadata from the device.
- **FR-AW-008**: The system shall provide a search field on the Results page that filters the analysis history by case-insensitive substring match on visible row data.
- **FR-AW-009**: The system shall allow the analysis history to be sorted by name, date, or status, with date descending as the default order.

2.1.4 FASTQ Processing and Classification

This subsection covers the analysis pipeline itself, from input ingestion through taxonomic classification and structured-output generation.

- FR-FP-001: The system shall accept FASTQ inputs in the .fastq, .fq, and .fastq.gz formats and shall reject unsupported file types at selection time.
- **FR-FP-002**: The system shall treat raw FASTQ files as read-only inputs throughout all stages of processing.
- **FR-FP-003**: The system shall execute the analysis as a Snakemake-orchestrated pipeline composed of independently maintainable rules.
- **FR-FP-004**: The system shall support a CPU classification path based on the local CLARK variant, executed inside a containerized runtime to ensure reproducibility.
- **FR-FP-005**: The system shall support a GPU classification path based on the GPU-accelerated CLARK variant, executed on a paired GPU device when available.
- **FR-FP-006**: The system shall use the user-selected execution path (CPU or GPU) for each analysis.

- **FR-FP-007**: The system shall split large FASTQ inputs into batches, classify each batch independently, and merge the per-batch abundance results into a single output.
- **FR-FP-008**: The system shall generate structured classification output (per-read assignments, abundance reports, and a JSON result document) consumable by the visualization and AI interpretation layers.
- **FR-FP-009**: The system shall report a clear failure status on the analysis when a pipeline stage cannot complete and shall preserve any partial logs for inspection.

2.1.5 Reference Database Management

Pathogenius ships with a default CLARK reference database and additionally allows the user to build and manage custom databases.

- **FR-DB-001**: The system shall display information about the default reference database, including the genome count and the index status.
- **FR-DB-002**: The system shall allow the user to create a custom CLARK database through a guided two-step wizard, taking a folder of genome files and a target-mapping file as input.
- **FR-DB-003**: The system shall allow the user to assign a name to a custom database.
- **FR-DB-004**: The system shall allow the user to optionally synchronize a newly created custom database to the paired GPU device at creation time.
- **FR-DB-005**: The system shall list all custom databases and allow each one to be deleted.
- **FR-DB-006**: The system shall allow the user to select which database (default or custom) is used for a given analysis at start time.

2.1.6 Result Visualization and Export

This subsection covers how completed analysis results are presented to the user and what export formats are supported.

- **FR-RV-001**: The system shall display an executive summary of detected organisms grouped by risk level.
- **FR-RV-002**: The system shall display species abundance as an interactive treemap.
- **FR-RV-003**: The system shall display the taxonomic hierarchy as an interactive sunburst chart.
- **FR-RV-004**: The system shall display read flow through the taxonomic hierarchy as an interactive Sankey diagram.
- **FR-RV-005**: The system shall display per-pathogen detail cards showing the assigned risk level, AMR genes, and virulence-gene information.
- **FR-RV-006**: The system shall display analysis quality metrics, including a radar visualization for the relevant metric set.
- **FR-RV-007**: The system shall apply a configurable confidence threshold to the displayed results and shall allow low-confidence results to be hidden.
- **FR-RV-008**: The system shall allow the user to export the result set as a JSON file.
- **FR-RV-009**: The system shall allow the user to export the result set as a structured PDF report containing the executive summary, quality metrics, the detected-organisms table, the per-pathogen detail cards, the AI summary when available, and a medical disclaimer.

2.1.7 AI-Generated Result Interpretation

Pathogenius runs a quantized medical language model on the host machine to produce a clinical-style summary from each analysis result.

- **FR-AI-001**: The system shall generate a clinical-style summary of the analysis using a quantized local language model running on the host machine.
- **FR-AI-002**: The summary shall cover the main findings, antimicrobial resistance concerns, suggested clinical action, and water-safety implications.
- **FR-AI-003**: The system shall stream the summary tokens into the results view as they are produced.
- **FR-AI-004**: The system shall display the summary as a dedicated AI Summary card within the result detail view.
- **FR-AI-005**: The system shall allow the user to enable or disable AI summary generation from the Settings page.
- **FR-AI-006**: The system shall perform all summarization inference on the local machine and shall not transmit any sample data, classification output, or prompt content to any external service.

2.1.8 Cloud Backup and Synchronization

Authenticated users can optionally back up their analyses to a managed cloud store and access them from any device on which they sign in. All backup data is encrypted on the host before upload, using the same per-user key that protects local at-rest data.

- **FR-CB-001**: The system shall allow an authenticated user to upload the result document of any completed analysis to the cloud from the Results page. The uploaded payload shall consist of the structured classification result together with its analysis name and sample type, and shall not include the raw FASTQ input, intermediate pipeline outputs, or the generated PDF report.
- **FR-CB-002**: The system shall encrypt the uploaded payload on the host using the user's per-account encryption key prior to upload, applying the same authenticated symmetric encryption used for local at-rest protection.
- **FR-CB-003**: The system shall store each encrypted payload under a per-user namespace and shall maintain corresponding metadata records sufficient to list the user's cloud-backed analyses.
- **FR-CB-004**: The system shall provide a Cloud Results view that lists all of the current user's cloud-backed analyses.
- **FR-CB-005**: Re-uploading a previously uploaded analysis shall overwrite the existing cloud copy rather than create a separate copy or block the operation.
- **FR-CB-006**: The system shall allow the user to download any of their cloud-backed analyses on the same or a different device using the same account credentials. Cross-device download is supported by re-deriving the encryption key from the user's password and the stored per-user salt.

- **FR-CB-007**: The system shall provide a dedicated cloud-delete action in the Cloud Results view that removes both the encrypted payload and its metadata record from the cloud. Deleting an analysis locally shall not, on its own, remove the cloud copy.
- **FR-CB-008**: The system shall surface upload, download, and cloud-delete failures to the user through dialog notifications, and shall leave local state unchanged on upload failure so that the user can retry.
- **FR-CB-009**: The system shall disable all cloud backup, listing, download, and cloud-delete functionality in Guest Mode.

2.1.9 Settings and Personalization

The Settings page exposes user-level preferences that affect analysis behavior, the user interface, notifications, and account security.

- **FR-ST-001**: The system shall allow the user to enable or disable AI summary generation.
- **FR-ST-002**: The system shall allow the user to set the confidence threshold used for filtering displayed results, with options of 0.5, 0.7, and 0.9.
- **FR-ST-003**: The system shall allow the user to toggle the visibility of low-confidence results.
- **FR-ST-004**: The system shall allow the user to enable or disable desktop completion notifications.
- **FR-ST-005**: The system shall allow the user to switch between dark and light themes, with the change applied immediately.
- **FR-ST-006**: The system shall allow the user to export the current settings to a JSON file.
- **FR-ST-007**: The system shall allow the user to import settings from a JSON file previously exported by the application.
- **FR-ST-008**: The system shall allow the user to reset all settings to their default values.
- **FR-ST-009**: The system shall persist settings locally and, when the user is authenticated, shall synchronize them to the user's account so that they are available on any device on which the user signs in.
- **FR-ST-010**: The system shall display read-only system information including the application version, the platform, and relevant runtime versions.

2.1.10 Real-Time Feedback and Notifications

This subsection covers the feedback the system gives the user during and after analyses, as well as the alert mechanisms used for errors and confirmations.

- **FR-NT-001**: The system shall display a running-analysis banner showing the analysis name, the current stage label, a status message, and a progress percentage.
- **FR-NT-002**: The system shall update the running-analysis banner in real time as the analysis progresses through pipeline stages.
- **FR-NT-003**: When the GPU execution path is in use, the system shall surface stage labels for the GPU-side phases (such as connecting, uploading, classifying, downloading, and finalizing).
- **FR-NT-004**: The system shall display global host telemetry on the Dashboard, including CPU utilization, RAM utilization, free disk space, and detected GPU information.

- **FR-NT-005**: The system shall raise a desktop notification when an analysis transitions to the completed, failed, or cancelled state, subject to the user's notification preference.
- **FR-NT-006**: The system shall display inline error feedback on the relevant form when login or registration fails, including specific cases such as wrong credentials, unverified email, suspended account, and validation failures.
- **FR-NT-007**: The system shall display dialog-based error notifications for operational failures, including invalid file selection, analysis startup failure, cloud upload or download failure, PDF export failure, database build failure, and administrative operation failure.
- **FR-NT-008**: The system shall display a transient confirmation indicator on the Settings page after a settings change is saved.

2.2 Non-Functional Requirements

2.2.1 Usability

- **NFR-USE-001**: The system shall be operable end to end through a graphical user interface, with no command-line interaction required for any user-facing task.
- **NFR-USE-002**: The system shall provide continuous visual feedback during long-running analyses through a progress indicator, a stage label, and a status message.
- **NFR-USE-003**: The system shall present classification results through interactive visualizations that support hovering, focusing, and inspection of individual taxa.
- **NFR-USE-004**: The system shall use consistent color coding to distinguish risk levels and confidence categories across views.
- **NFR-USE-005**: The system shall support a dark theme and a light theme, with the user's choice applied immediately and persisted across sessions.
- **NFR-USE-006**: The system shall organize functionality into a small number of clearly labeled top-level areas (Dashboard, New Analysis, Results, Database, Settings, and Administration when applicable) to support navigation by users without bioinformatics expertise.

2.2.2 Reliability

- **NFR-REL-001**: The full analysis path, from FASTQ ingestion through classification and AI summarization on the local execution path, shall be operable without an internet connection.
- **NFR-REL-002**: The local execution path shall be operable on the host machine without runtime dependencies on external services for analysis, classification, or AI summarization. The components required for offline operation are the analysis tools (Snakemake, the containerized CLARK runtime), the default reference database, and the local language model.
- **NFR-REL-003**: The pipeline shall be deterministic in the sense that identical inputs and reference data produce identical classification output.
- **NFR-REL-004**: Raw FASTQ inputs shall remain unmodified throughout the pipeline.
- **NFR-REL-005**: Cancelling an analysis shall reliably terminate the associated process tree on the host platform.
- **NFR-REL-006**: The system shall persistently report the terminal state of every analysis (completed, failed, or cancelled) so that the user can distinguish between them after the fact.
- **NFR-REL-007**: Cloud-stored results shall be protected by authenticated encryption so that any tampering with the stored ciphertext is detected at decryption time.

2.2.3 Performance

- **NFR-PER-001**: The local analysis pipeline shall be capable of running on mid-range consumer laptops without dedicated computing infrastructure.
- **NFR-PER-002**: The classification stage shall use multi-threaded CPU processing on the local execution path.
- **NFR-PER-003**: The system shall split large FASTQ inputs into smaller batches and merge their results to keep memory usage within the available host budget.
- **NFR-PER-004**: When a paired GPU device is available and selected by the user, the system shall be capable of executing the same analysis using the GPU-accelerated CLARK variant for higher throughput.
- **NFR-PER-005**: The AI summarization layer shall operate within a bounded context size and a bounded output length so that interpretation completes within a predictable time budget on a laptop CPU.
- **NFR-PER-006**: The system shall detect host hardware (CPU, RAM, GPU presence) at startup and shall present this information on the Dashboard.

2.2.4 Maintainability

- **NFR-MAI-001**: The system shall be structured as four modular layers (Frontend, Backend, Workflow, AI / Interpretation), each with a clearly defined responsibility and a stable interface to its neighbors.
- **NFR-MAI-002**: The analysis workflow shall be implemented as discrete Snakemake rules so that individual stages can be modified, replaced, or extended without affecting the rest of the system.
- **NFR-MAI-003**: The desktop application shall use standard Electron process separation between the main process, a sandboxed preload, and the renderer.
- **NFR-MAI-004**: The codebase shall be version-controlled and shall follow consistent coding conventions across files and modules.
- **NFR-MAI-005**: The system shall produce persistent session data and structured logs sufficient to support debugging of an analysis after the fact.
- **NFR-MAI-006**: User settings shall be exportable to and importable from a portable file format to support reproducibility, support requests, and migration between machines.

2.2.5 Security

- **NFR-SEC-001**: The system shall require authentication before granting access to features that operate on a registered user's account, including the registered user's analysis history, encrypted local storage, cloud backup, account-security operations such as password change, and administrative functions. Guest Mode shall not grant access to any of the above.
- **NFR-SEC-002**: The system shall require email verification before a newly registered account is permitted to log in.
- **NFR-SEC-003**: Analysis results stored on disk for authenticated users shall be encrypted at rest using authenticated symmetric encryption.

- **NFR-SEC-004**: The encryption key used for at-rest protection shall be derived from the user's password through a salted, iterated key-derivation function with a per-user random salt.
- **NFR-SEC-005**: The encryption key shall be held in process memory only, shall never be written to disk, and shall be cleared on logout.
- **NFR-SEC-006**: Authentication tokens shall be persisted through the operating system keychain rather than in plain configuration files.
- **NFR-SEC-007**: Role-based access control shall be enforced at both the application layer and the backing database's security rules.
- **NFR-SEC-008**: The renderer process shall be sandboxed and isolated from Node.js APIs, and all communication between the renderer and the main process shall pass through a controlled bridge.
- **NFR-SEC-009**: The system shall validate inputs that flow into file-system paths against expected formats and shall sanitize inputs that flow into shell commands. The file-selection dialog shall restrict input to supported FASTQ extensions.
- **NFR-SEC-010**: The system shall not transmit sample data, classification results, or AI summarization content to any external service except where the user has explicitly initiated a cloud backup of an analysis result. For cloud backup, the payload shall be encrypted on the host prior to upload using the user's per-account encryption key. The AI-generated clinical summary shall not be included in any cloud transmission and shall remain on the host on which it was generated.
- **NFR-SEC-011**: Analysis results created in Guest Mode shall be stored on disk in unencrypted form, by design, to support immediate offline use without a derived encryption key. Users requiring at-rest protection of analysis results shall use a registered account.

2.3 Pseudo Requirements

Pseudo requirements capture the binding implementation decisions adopted at the project level. They constrain how the functional and non-functional requirements above are realized and remain in force for any future work on the system.

- **PR-01**: The desktop application shall be built using Electron.js, with standard process separation between the main process, a sandboxed preload, and the renderer.
- **PR-02**: The analysis pipeline shall be orchestrated using Snakemake, with each pipeline stage implemented as a discrete rule.
- **PR-03**: Taxonomic classification shall be performed using the CLARK family of classifiers, with a CPU variant for the local path and a GPU-accelerated variant for the paired-device path.
- **PR-04**: The local CLARK runtime shall be executed inside a containerized environment to ensure reproducibility across host platforms.
- **PR-05**: AI-generated result interpretation shall be performed by a quantized open-source medical language model executed locally through a native inference runtime, with no remote inference path.
- **PR-06**: User authentication and role storage shall be backed by a managed authentication and database service, with security rules enforced server-side independently of the client.

- **PR-07**: Cloud backup of analysis results shall use a managed object-storage service, and all uploaded blobs shall be encrypted on the host before transmission.
- **PR-08**: At-rest encryption shall use an authenticated symmetric cipher with per-user keys derived from user credentials through a salted, iterated key-derivation function.
- **PR-09**: The project shall be developed under Git version control, with the source repository hosted on GitHub.

3 Final Architecture and Design Details

3.1 Overview

The Pathogenius system is built using a four-layer modular architecture that transforms raw long-read FASTQ data into species-level pathogen identifications through an automated, locally orchestrated workflow. The architecture separates user interaction (Frontend Layer), orchestration and platform integration (Backend Layer), computational analysis (Workflow Layer), and result interpretation (AI / Interpretation Layer) into distinct subsystems with well-defined responsibilities and stable interfaces. Persistent data and reference data sit beneath these layers as shared artifacts. This separation supports maintainability, testability, and incremental evolution of any one layer without requiring changes to the others.

The high-level deployment view of the system is shown in Figure 1. Two deployment zones are involved. The Host Machine, an Electron-based desktop application running on a consumer-grade laptop or workstation, hosts the four architectural layers. When the host is equipped with a CUDA-compatible NVIDIA GPU, the GPU-accelerated classification path runs on the GPU-enabled machine, either the host or the connected device, alongside the rest of the application. A set of managed Cloud Services (Firebase Authentication, Firestore, Firebase Storage) supports optional account-related and synchronization features for registered users.

Communication between the Frontend and Backend layers uses Electron's IPC mechanism, exposed to the renderer through a controlled contextBridge as a structured window.api namespace. Communication between the Backend and the Workflow Layer uses a Snakemake child process spawned by the Backend, with progress information parsed back from the workflow's standard output. Communication between the Backend and the AI / Interpretation Layer is in-process at deployment time, since the language model runs inside the Backend's main process; the architectural separation reflects responsibility, not deployment boundaries. Communication between the Backend and the Cloud Services uses HTTPS REST calls. The GPU-accelerated classification path, when present, is invoked locally on the host and introduces no additional network communication. The local execution path, comprising Frontend, Backend, the Workflow Layer (CPU or GPU branch), and the AI / Interpretation Layer, operates without an internet connection once the application and its supporting components have been installed. Cloud Services are invoked only when the user has chosen to register an account and explicitly opts into features such as cloud backup. Guest Mode, which requires no account and no network, exercises only the local execution path.

3.2 Subsystem Decomposition

This subsection describes the internal structure of each of the four subsystems and the components that make them up. The Frontend Subsystem provides the UI; the Backend Subsystem provides orchestration and platform services; the Workflow Subsystem performs the computational analysis; and the AI / Interpretation Subsystem produces the human-readable summary of analysis results.

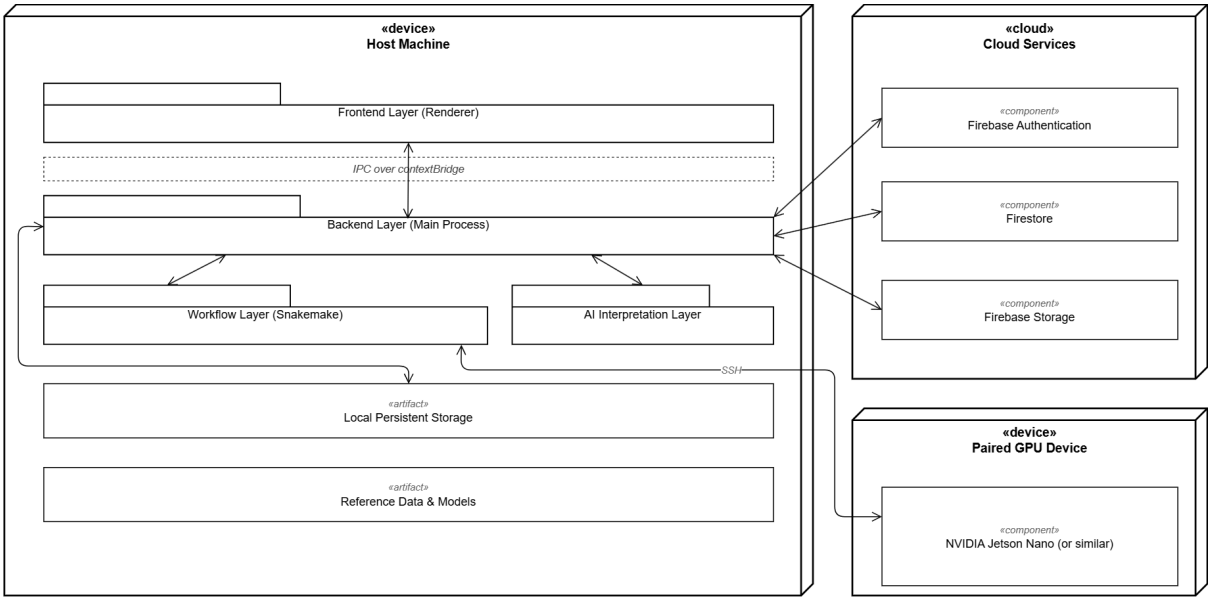

Figure 1. *High-level deployment architecture of Pathogenius.*

3.2.1 Frontend Subsystem

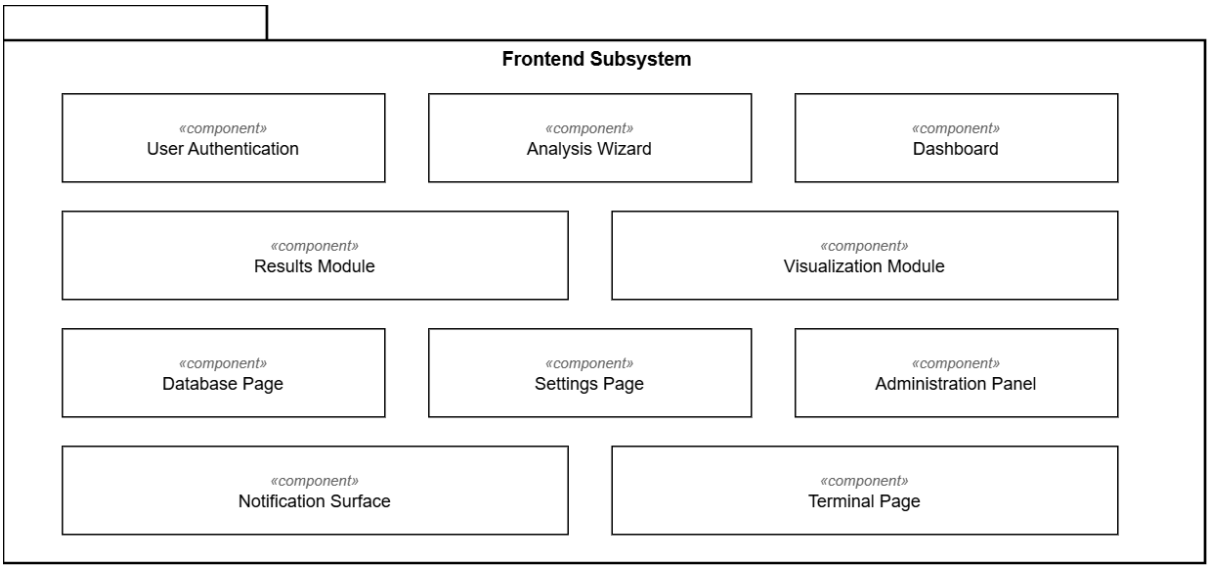

Figure 2. *Internal structure of the Frontend Subsystem of Pathogenius.*

The Frontend Subsystem is a desktop user interface implemented as the Electron renderer process. It provides all user-facing functionality without requiring command-line interaction at any point. The internal structure of the Frontend Subsystem is shown in Figure 2.

The User Authentication Module covers all account-related interactions, including registration with email verification, login with credential validation, password reset via email, password change from within the session, and entry into Guest Mode. Each flow is implemented as its own page or modal, with inline form-level error feedback for predictable error cases.

The Analysis Wizard guides the user through starting a new analysis as a three-step flow: file selection (FASTQ inputs are validated against supported extensions, with SeqKit statistics computed for each selected file), engine and database configuration (CPU or GPU branch, default or custom reference database, optional batch processing), and a review-and-launch step that summarizes the configuration before submission.

The Dashboard provides system overview information at a glance: live host telemetry (CPU and RAM utilization, free disk space, detected GPU information), a list of recently completed analyses, and a banner that surfaces the progress of any running analysis. The Dashboard polls the Backend periodically to keep its telemetry current.

The Results Module is the central hub for all analysis outcomes. It is organized as three tabs: a Running Analyses tab that shows in-flight analyses with progress indicators and pause / resume / cancel controls; a Local Results tab that lists completed, failed, and cancelled analyses for the current user; and a Cloud Results tab that lists analyses backed up to the cloud. The Results Module also renders the result-detail view, in which an individual analysis is presented through its visualizations, the AI summary card, and an exportable report.

The Visualization Module renders the four chart types used in the result-detail view: a treemap of species abundance, a sunburst of the taxonomy hierarchy, a Sankey diagram of read flow through the taxonomy, and a radar chart of pathogen attributes. It also renders per-pathogen detail cards displaying risk level, antimicrobial-resistance gene information, and virulence-gene information. All charts share a common tooltip system and are rendered as inline SVG without external charting dependencies.

The Database Page exposes the reference-database management interface. It displays the status of the default CLARK reference database (genome count, index status), provides a two-step wizard for creating custom databases from a folder of genome files and a target-mapping file, lists previously created custom databases, and offers an optional toggle for synchronizing a newly created custom database to the paired GPU device.

The Settings Page exposes user-level preferences in nine grouped categories, covering account security (visible to registered users only), AI summary generation, analysis confidence thresholds and low-confidence filtering, desktop completion notifications, dark / light theme, settings export and import, an administration panel (visible to administrators only), system information, and contact information.

The Administration Panel, conditionally exposed within the Settings Page for administrator accounts, lists registered users with a search field and provides per-user actions including account suspension, reactivation, role assignment, password-reset triggering, and account deletion.

The Notification Surface is a coordinated set of feedback mechanisms rather than a single component. Long-running analyses surface progress through banners on both the Dashboard and the Results page. Analysis state transitions (completion, failure, cancellation) raise OS-level desktop notifications when the user has enabled them. Operational failures (invalid file selection, network errors, administrative-action failures) surface as modal dialogs. Form-level validation failures surface as inline error text on the relevant form. A transient banner confirms saved changes on the Settings page.

The Terminal Page provides a live console relay from the Backend process, intended primarily as a debugging and operational-monitoring aid. It displays log lines as they are produced by the Backend and the Workflow.

3.2.2 Backend Subsystem

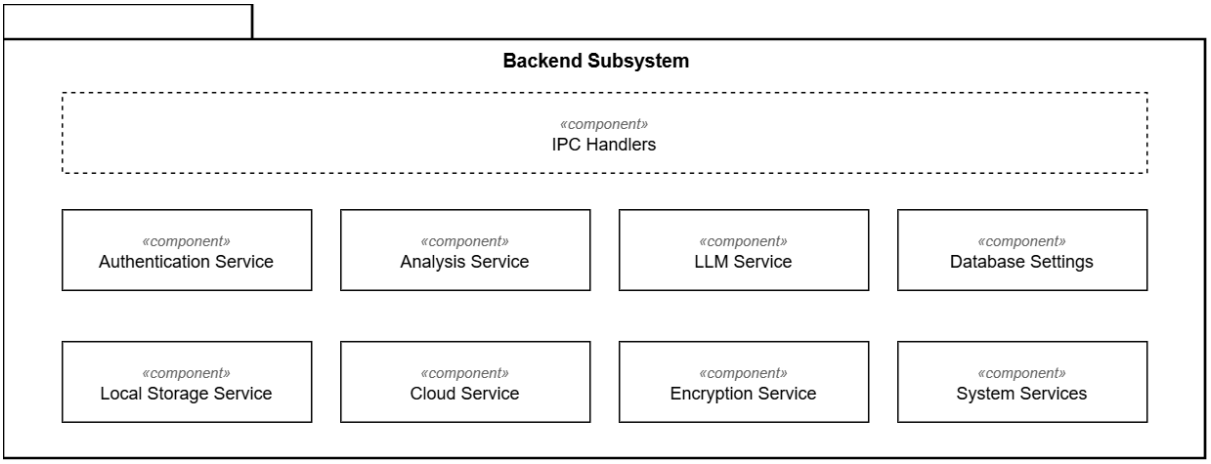

Figure 3. *Internal structure of the Backend Subsystem of Pathogenius.*

The Backend Subsystem is implemented as the Electron main process and serves as the central orchestrator of the system. It receives all requests from the Frontend through IPC handlers, delegates to the Workflow Subsystem for computational tasks, handles all communication with external Cloud Services, and manages local persistent state. The internal structure of the Backend Subsystem is shown in Figure 3.

The IPC Handlers are the entry surface of the Backend. They are registered on Electron's IPC channels in the main process and are organized by functional area: authentication, settings, file system, analysis, database, cloud, administration, language model, and system. The renderer accesses these handlers through a controlled window.api namespace that is exposed by the preload script via Electron's contextBridge mechanism. Handlers route requests to the appropriate domain service, validate parameters where applicable, and return structured responses back to the renderer.

The Authentication Service implements all account-related logic by interacting with the Firebase Authentication REST API and the Firestore document model. It handles registration (account creation, email verification dispatch, initial profile creation, encryption-key material initialization), login (credential validation, profile retrieval, session establishment, refresh-token persistence to the OS keychain), session restoration on startup, password reset and password change flows, and administrative operations on user profiles. It also handles the synchronization of user settings to and from Firestore for registered users.

The Analysis Service is responsible for the lifecycle of an analysis from launch through completion. It generates per-analysis configuration, creates the working directory and ownership records, spawns Snakemake as a child process, parses Snakemake's standard-output stream into structured progress events that are forwarded to the renderer, monitors process state, and supports cancellation through cross-platform process-tree termination. It also coordinates the optional batch-processing path, in which large inputs are split with SeqKit, classified independently, and merged at the abundance stage. After successful completion, it merges the workflow's structured output with frontend-side metadata to produce the final result document.

The LLM Service runs the local language model in-process within the Backend, using node-llama-cpp as the inference runtime. A persistent model context is created once at load time and reused across calls; per-request sequence slots are obtained for each generation and disposed after each call, keeping memory usage predictable. The service exposes both blocking and streaming inference paths to the IPC layer; the streaming path forwards each generated token to the renderer as it is produced, which the Frontend uses to populate the AI Summary card progressively.

The Database Settings Service maintains the registry of custom reference databases (name, location, status, optional GPU-sync flag) and provides the database-build entry point that runs the CLARK index-construction script as a child process. It also exposes the default-database information used by the Database Page on the frontend.

The Local Storage Service implements the encrypted at-rest store for registered users. It writes each completed analysis result as two files: an encrypted blob containing the structured result and a plain metadata file containing the fields needed to render the analysis-history list (status, timestamps, detected count, cloud-backup flag). This split allows the history view to be rendered without decrypting any results.

The Cloud Service implements the optional cloud-backup path. It encrypts the result document on the host, uploads the resulting ciphertext to Firebase Storage under the user's namespace, and writes corresponding metadata to Firestore so that the cloud-backed result list can be enumerated without decryption. It also supports cross-device download (which decrypts using the per-user key derived locally from the user's password) and explicit cloud-side deletion of a backup.

The Encryption Service is a shared cryptographic dependency consumed by both the Local Storage Service and the Cloud Service. It performs key derivation through PBKDF2-SHA-512 with a per-user random salt, holds the derived 32-byte AES key in process memory for the duration of the session, and

provides authenticated encryption and decryption using AES-256-GCM. The derived key is never written to disk and is cleared from memory on logout.

The System Services collect a number of cross-cutting platform concerns: hardware detection, in which GPU information is probed once at startup using systeminformation and CPU and RAM utilization are sampled on a polling interval for the Dashboard telemetry; PDF generation, in which the renderer builds the report's HTML and the main process renders it to PDF using a hidden BrowserWindow and webContents.printToPDF; per-analysis ownership tracking via on-disk owner records; guest-mode cleanup on explicit logout; and the relay of console output to the Frontend's Terminal Page.

3.2.3 Workflow Subsystem

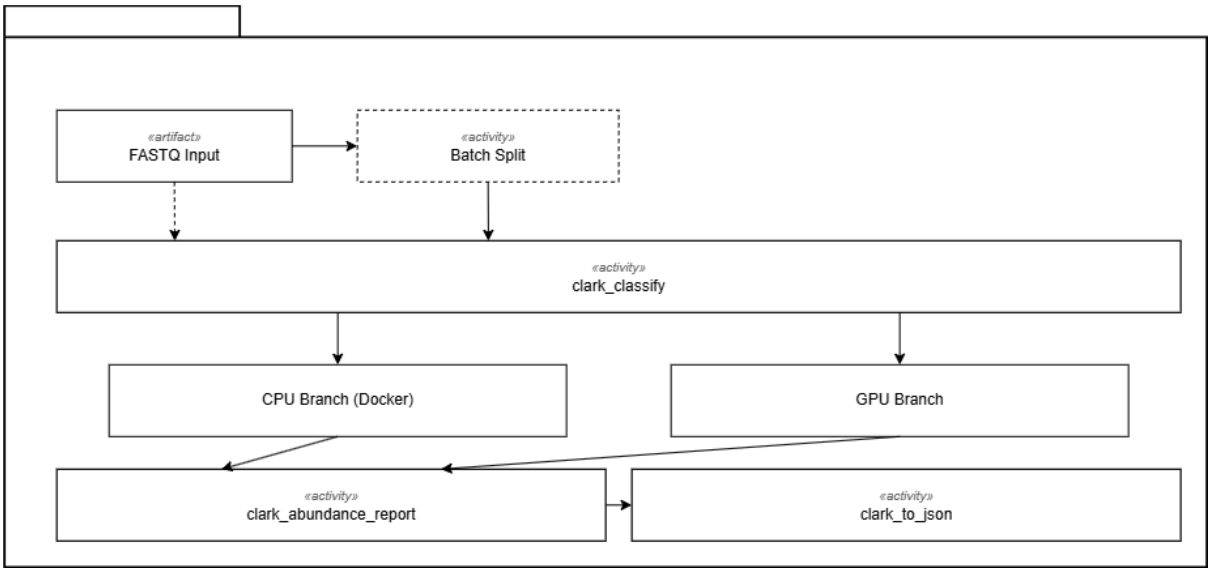

Figure 4. *Internal structure of the Workflow Subsystem of Pathogenius.*

The Workflow Subsystem is the computational core of the system. It is implemented as a Snakemake-orchestrated pipeline in which each stage is an independently testable rule. The Backend launches the workflow as a single Snakemake child process and consumes its progress reports via standard output. The internal structure of the Workflow Subsystem is shown in Figure 4.

When the user has enabled batch processing in the Analysis Wizard, the input FASTQ file is first split into two equal parts using SeqKit. Each part is classified independently and the per-part results are merged at the abundance stage. When batch processing is disabled, the input FASTQ file is consumed directly by the classification step.

The clark_lite_classify rule performs the central taxonomic classification. It is implemented as a single Snakemake rule with two execution branches selected at runtime based on the user's engine choice. The CPU branch runs the CLARK-l classifier inside a Docker container on the host machine, with the reference database, FASTQ input, and result directory mounted as container volumes. The GPU branch runs CuCLARK-l on an accessible CUDA-compatible GPU, including a pre-flight check of available GPU memory, launch of the classification process, periodic polling for completion, and

cleanup of intermediate outputs. Both branches produce the same output: a per-read classification CSV that is consumed identically by downstream rules.

The clark_abundance_report rule computes per-species abundance proportions and absolute read counts from the classification CSV. The CPU branch uses CLARK's bundled abundance estimator. The GPU branch uses a local Python implementation that reads the classification output, joins it against a locally cached copy of the NCBI taxonomy, and emits the abundance report in the same format as the CPU branch.

The clark_to_json rule converts the abundance report into the structured JSON document consumed by the Frontend. It assigns a numeric confidence score and a categorical risk level to each detected pathogen, computes summary statistics (total reads, classified reads, species detected, antimicrobial-resistance signals), and produces the taxonomy composition breakdown used by the Visualization Module. This rule runs locally for both execution branches.

After the workflow completes, the Backend's Analysis Service performs a final post-processing step that merges the workflow's JSON output with frontend-side metadata (analysis name, sample type, classifier label, completion timestamp) to produce the result document persisted under the analysis directory and consumed by the Frontend.

3.2.4 AI / Interpretation Subsystem

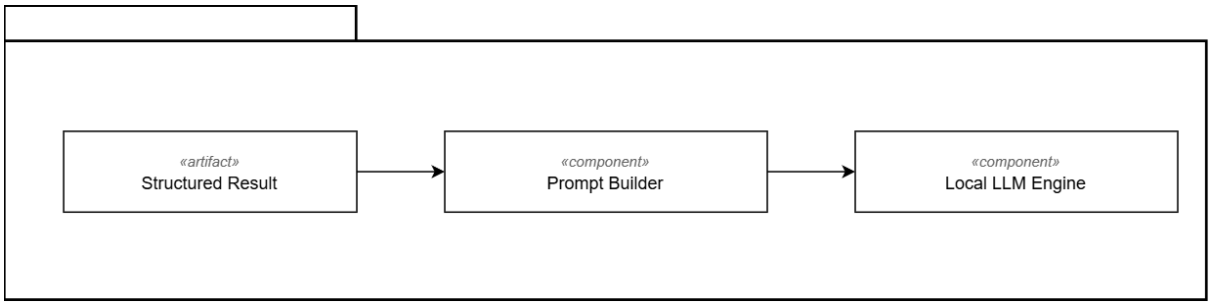

Figure 5. Internal structure of the AI / Interpretation Subsystem *of Pathogenius.*

The AI / Interpretation Subsystem produces a clinical-style natural-language summary of an analysis result. It runs entirely on the host machine with no external inference path. The internal structure of the AI / Interpretation Subsystem is shown in Figure 5. The subsystem is invoked by the Frontend after a result has been opened in the result-detail view, provided that the user has enabled local AI generation in the Settings page. Its input is the structured result document produced by the Workflow Subsystem and post-processed by the Backend.

The Prompt Builder constructs a single text prompt from the structured result. It injects the analysis metadata (analysis name, sample type) and summary statistics (total reads, classification rate, species detected, AMR signals, average quality), and embeds a structured per-pathogen list (name, strain, abundance, confidence, risk level, AMR genes, virulence genes). The prompt also contains instructions that direct the model to produce four paragraphs covering the main findings, antimicrobial-resistance

concerns, suggested clinical action, and water-safety implications, in plain text without markdown formatting.

The Local LLM Engine runs a quantized medical language model on the host through the node-llama-cpp inference runtime, which is loaded into the Backend's main process. A persistent model context is created at first use and retained across calls to avoid repeated initialization cost. Each generation acquires a fresh sequence slot from the context, runs to a bounded output length, and releases the slot on completion. As the model produces tokens, the LLM Service forwards each token to the Frontend through a dedicated IPC channel, allowing the AI Summary card in the result-detail view to render the summary progressively rather than waiting for completion.

3.3 Hardware and Software Mapping

Pathogenius is designed to run on a consumer-grade laptop or workstation as the primary host. The Electron application is built against current versions of the framework and is structurally portable across Windows, macOS, and Linux. The CPU classification branch requires Docker (Docker Desktop on Windows and macOS, Docker Engine on Linux) to host the CLARK runtime container; this is the only required external runtime beyond the Electron application itself.

The host computational profile assumes a multi-core CPU and sufficient RAM to host the Electron process, the Snakemake child process, the CLARK Docker container, and the in-process language model concurrently. Hardware detection at startup probes the available CPU, RAM, and GPU information using the system information library and reports it on the Dashboard. The detected information also informs the Frontend's display of the available execution-engine options.

The optional GPU classification path runs on a CUDA-compatible NVIDIA GPU accessible by the host machine. When such a GPU is present and selected by the user, the GPU-accelerated CuCLARK-l variant is invoked either locally using the localhost or can be operated with a SSH connection to the GPU-enabled device, the system handles all necessary file transfers and database creations in the target device. GPU-accelerated pipeline operates against the same reference database used by the CPU branch. A discrete GPU is not required for system operation; users without one run the system entirely on the CPU branch.

External cloud services are required only for features that cross host-machine boundaries: account-based authentication, settings synchronization across devices, and cloud backup of analysis results. These features are implemented against Firebase Authentication, Cloud Firestore, and Firebase Storage, all reached over HTTPS. Guest Mode and the local execution path do not invoke any of these services.

The hardware/software mapping in Table 1 describes how the five architectural layers are deployed onto available hardware resources, and how that deployment differs in the host-only and edge-attached configurations.

Table 1: Final hardware/software mapping for Pathogenius.

Hardware Component	Software Deployed	Description
Host CPU	Electron.js Frontend, Backend API, Snakemake engine, seqkit, CLARK-lite, LLM	Primary compute resource. Dynamically scales threads based on detected cores. Executes all layers if no GPU/Edge is present.
Host RAM	CLARK k-mer index, Snakemake working memory, AI model weights, Firebase SDK	Managed via user-configurable memory caps. Classifier and LLM are scheduled sequentially to prevent VRAM/RAM overflow.
NVIDIA GPU (Optional)	CuCLARK classifier	Used for either classification or AI inference (not simultaneously). System auto-detects and falls back to CPU if unavailable.
Local Disk	FASTQ inputs, FASTA references, CLARK indices, Analysis history, AI weights	Single user-configurable directory. FASTQ inputs are read-only. Indices are versioned for multi-revision coexistence.
Jetson Nano (Optional)	Snakemake worker, seqkit, Database Builder, CuCLARK / CLARK-lite, Edge Mirror	Workflow Layer dispatches tasks via Edge Execution Adapter. Only structured outputs return to host; raw data is purged after run.

Network (LAN)	Host ↔ Jetson Nano control channel	Used only for edge execution. Facilitates artifact transfer and remote execution commands. No internet required.
Firebase (Optional)	Cloud Firestore, Cloud Storage, Firebase Authentication	Off-path storage used exclusively for report sync (JSON/PDF/Metadata). Never handles raw sequencing data.

3.4 Persistent Data Management

Pathogenius writes several distinct categories of persistent state during operation, each managed in a manner appropriate to its sensitivity, lifetime, and access pattern.

Per-analysis working directories are created at analysis start and hold the analysis configuration, the file recording the owning user identity, the workflow's intermediate outputs, and on successful completion the final result document together with the classification and abundance CSVs. These directories live in a working area within the application's installation footprint. They are produced by both registered users and Guest Mode sessions, but only registered users have their final results promoted to the encrypted at-rest store described below; Guest Mode results remain in the working directories until explicit logout, at which point they are removed by the Backend's guest-cleanup routine.

The encrypted at-rest store is the long-term home of completed analyses for registered users. It lives under the operating system's per-user application data directory, partitioned by user identity. Each analysis is stored as two files: an encrypted blob (.enc) containing the structured result document, and a plain metadata file (.meta.json) containing the fields required to render the analysis-history list without decrypting the result. The encryption is performed by the Encryption Service described in Section 3.5. The split between encrypted payload and plain metadata is deliberate: it allows the Frontend to enumerate the user's analysis history at login without unlocking any encryption material, deferring decryption to the moment a specific result is opened.

Reference data consists of the default CLARK reference database and any custom databases the user has created. The default database is expected at a known location within the application's installation footprint and is provided as a one-time setup deliverable rather than being downloaded at runtime. Custom databases are stored at user-selected paths and are tracked in a machine-scoped registry that lists each database by name along with its build status and optional GPU-sync flag. Because the registry is machine-scoped rather than user-scoped, custom databases on a given host are visible to all

users of that host; this matches the system's intended use as a single-laboratory or shared-workstation tool.

User settings have a two-tier storage model. For registered users, settings are persisted to Firestore on every change so that they are available on any device on which the user signs in. As a fallback for Guest Mode and for offline operation, settings are also written to the renderer's local storage. Settings changes made in a session are written to local storage immediately and persisted to Firestore when a session is available.

A small number of caches reduce repeated work. The hardware-detection result is cached in memory at startup and reused for subsequent dashboard requests. HTML templates loaded by the renderer are cached in memory for the lifetime of the application. NCBI taxonomy data downloaded from the GPU device on first GPU-path execution is cached in the workflow area for reuse on subsequent GPU-path runs.

Logs are not persisted to disk as a long-term record. The Backend's console output is relayed to the Frontend's Terminal Page in real time for debugging and monitoring purposes, but no rotating log file is maintained on the host. Per-analysis state (including failure reason, when applicable) is preserved in the analysis's metadata file so that the user-facing history view remains consistent with any later inspection.

3.5 Access Control and Security

Authentication is required before access to any account-bound functionality, including the registered user's analysis history, encrypted local storage, cloud backup, account-security operations, and the Administration Panel. Authentication is implemented against the Firebase Authentication REST API. Account creation requires email verification before the account becomes able to log in. The login flow validates credentials, retrieves the user's profile and encryption-key material from Firestore, derives the at-rest encryption key, and persists the resulting refresh token to the operating system keychain via the platform-native credential store. Session restoration on subsequent application launches uses the refresh token to obtain a new session without prompting the user for credentials. Logout clears all in-memory session state and removes the refresh token from the keychain.

Guest Mode is supported as an unauthenticated, immediate-access path for offline use. It does not grant access to any of the account-bound functionality listed above. A Guest session can run analyses, view results from the same session, and adjust non-account-related settings. Guest Mode data is cleared automatically when the session ends.

Role-based access control distinguishes regular users from administrators. The role assignment is held in the user's Firestore profile document. Administrative privileges are enforced at two points: at the application layer, where each administrative IPC handler in the Backend explicitly checks the active user's role before acting; and at the Firestore security-rules layer, where role-changing and status-changing writes are restricted server-side. Holding the role in Firestore (rather than inferring it locally) means that role changes take effect immediately on the user's next request rather than requiring a re-login.

At-rest protection of analysis results for registered users is provided by the Encryption Service. The service derives a 32-byte symmetric key from the user's password through PBKDF2-SHA-512, using a per-user random salt persisted in the user's Firestore profile. The derived key is held in process memory for the duration of the session, never written to disk, and cleared from memory on logout or process exit. Encryption uses AES-256-GCM, an authenticated mode whose authentication tag is verified on each decryption; tampering with a stored ciphertext therefore produces a decryption failure rather than a silently incorrect result. The same key is used by the Cloud Service to encrypt result payloads on the host before upload, so that a user signing in on a second device with the same credentials can re-derive the key locally and decrypt their cloud-stored results.

The Electron desktop shell is configured following the framework's recommended hardening posture. The renderer process runs with contextIsolation enabled, nodeIntegration disabled, and the sandbox enabled, so that the renderer cannot reach Node.js APIs directly. All renderer-to-main communication is mediated by the contextBridge as a controlled window.api namespace whose methods are explicitly registered in the preload script. There is no direct exposure of file-system, child-process, or operating-system APIs to renderer code.

Inputs that flow into file-system paths or external command invocations are validated before use. Analysis identifiers used to construct on-disk paths must match a fixed regular-expression pattern before the Backend will perform any read, write, or delete on the corresponding directory. The file-selection dialog restricts user input to FASTQ extensions before a file is admitted to the analysis pipeline. Sample names that participate in workflow command construction are sanitized prior to invocation.

Network traffic outside the local host is limited to the cloud services described in Section 3.3. Communication with Firebase Authentication, Firestore, and Firebase Storage uses HTTPS. The GPU classification path, when used, runs either on the GPU-enabled device (host or a separate device). Sample data, classification results, and AI-generated summaries are not transmitted to any external service except where the user has explicitly initiated a cloud backup of an analysis result, in which case the payload is encrypted on the host before transmission. The AI-generated clinical summary, in particular, is never included in any cloud transmission and remains on the host on which it was generated.

4 Development/Implementation Details

Pathogenius is implemented as an Electron desktop application whose main process orchestrates a Snakemake-driven bioinformatics workflow, a local large-language-model service, a Firebase-backed cloud-sync layer, and an optional Jetson Nano edge node reachable over SSH. The repository is split into two top-level packages: Patho-genius/, which holds the Snakemake workflow and the universal database builder, and frontend/, which holds the Electron application together with all the services that drive analyses, encryption, authentication, and visualization. This section describes how each subsystem was built, the technology choices we made, and the development practices that kept five contributors working in parallel.

4.1 Frontend

The Pathogenius frontend is implemented as an Electron desktop application using vanilla JavaScript, HTML templates, and a modular CSS architecture. We chose Electron because it lets the application ship as a single self-contained installer while still giving us direct access to the local filesystem, the OS keychain, the long-running Snakemake workflow process, and a local LLM runtime, all of which are essential for an offline-capable platform that has to host its own bioinformatics pipeline. We chose vanilla JavaScript over a framework like React for two reasons: the application is screen-driven rather than data-driven, so the cost of a virtual DOM and a build pipeline did not pay for itself; and keeping the renderer dependency-free made it easier for five contributors with different IDEs and operating systems to touch the same files without churn over tooling.

The application follows Electron's standard two-process split. The main process (frontend/src/main/main.js) owns the long-lived state: it creates the application window, registers IPC handlers for every privileged operation, supervises Snakemake when an analysis runs, and holds the keychain handles used by the auth service. The renderer process is sandboxed Chromium that runs the user interface; it has no direct access to Node APIs and reaches the main process only through a narrow set of namespaced bridges installed by the preload script.

Figure 9 — Frontend Process Topology and IPC Bridge

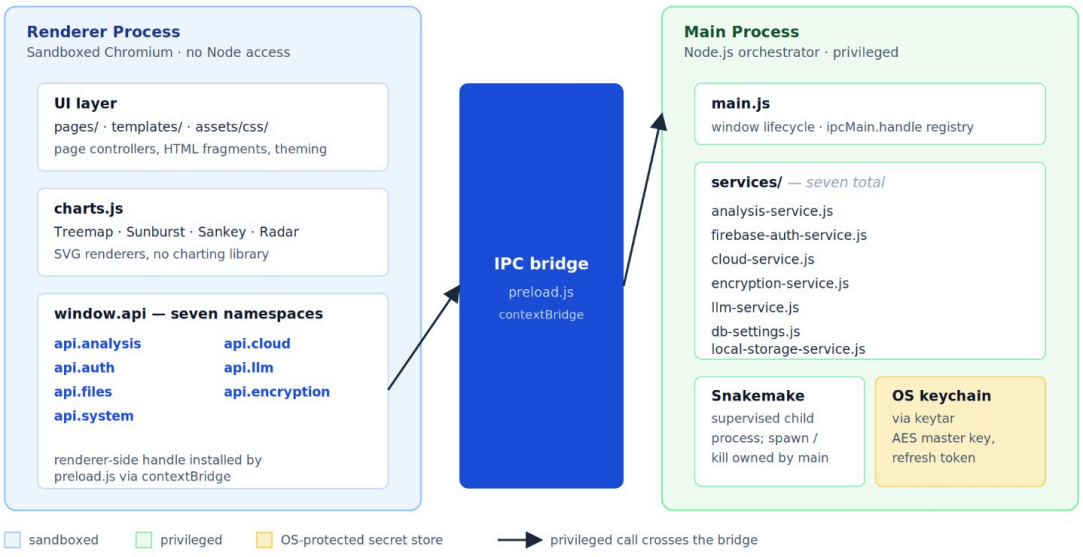

Figure 9: Frontend process topology. The renderer hosts the HTML templates, the page controllers, and the SVG chart renderers; the main process owns the IPC bridge, the seven services, and the supervised Snakemake child process.

4.1.1 Renderer Layout and Navigation

The renderer is structured around three concerns kept in separate directories: assets/ holds the modular CSS (variables, base, components, charts, layout) and local Inter font files; templates/ holds reusable HTML fragments (sidebar, modals, and one template per page); and pages/ holds the JavaScript controller that owns each page's behaviour. A single template-loader.js module is responsible for fetching templates on demand and mounting them into the page's content slot, which is what makes navigation feel instant after the initial app load.

Global navigation state lives in app.js. It owns tab switching, theme management, and the sidebar's active-page indicator. Page-specific behaviour lives in the matching pages/ controller, login.js, register.js, dashboard.js, newanalysis.js, results.js, database.js, and settings.js, and each controller is loaded once when its page is first opened. The sidebar template is shared, so the same navigation chrome wraps every page.

The New Analysis screen (newanalysis.js) is implemented as a single-page configuration form. The user picks a FASTQ file through the native file dialog (api.files.selectFile()), selects the engine (CPU or GPU), enters an analysis name and sample type, and starts the analysis. The Results screen (results.js) is the most behaviour-heavy controller: it owns the analysis history list, the detail view with pathogen cards, the four SVG chart renderers (described in Section 4.1.4), the AI-summary trigger that calls the LLM service, the cloud-sync controls, and the export buttons.

4.1.2 Styling and Theme

Visual design uses a centralized theme based on CSS custom properties declared in assets/css. A fixed colour palette, a single typographic scale, and a small set of shared layout primitives (cards, tables, status pills, progress bars, confidence chips) are reused across every page so that confidence indicators, status states, and notification severities look identical wherever they appear. Theme switching (dark / light) is driven by toggling a root-level data attribute, with the actual colour values resolved through the CSS variables, no JavaScript-driven restyling is needed. The application defaults to a dark theme since field laptops are often used in poor lighting conditions where a dark background reduces glare.

4.1.3 Preload Bridge and IPC

The preload script (frontend/src/main/preload.js) is the only place where the renderer and the main process touch each other. It uses Electron's contextBridge.exposeInMainWorld to expose seven namespaced bridges on the renderer's window.api object: api.analysis, api.auth, api.files, api.system, api.cloud, api.llm, and api.encryption. Each method on these bridges corresponds to an IPC handler registered in main.js. A typical call looks like await api.analysis.start(config) on the renderer side and an ipcMain.handle('analysis:start', ...) implementation on the main side; the bridge keeps every privileged operation routed through the same audit point.

Long-running operations report progress back through a separate IPC channel rather than over a return value. The main process emits progress events as Snakemake produces output, the preload script

forwards them to the renderer, and the New Analysis page subscribes to them while a run is active. Because there is only one renderer in this application, we did not need a WebSocket server or any other in-process IPC abstraction, Electron's built-in channels are sufficient.

4.1.4 Visualization Suite

The four interactive charts on the Results page, Treemap, Sunburst, Sankey, and Radar, are implemented in charts.js as SVG renderers. We did not reach for a charting library; the four chart types we needed were specific enough that hand-rolling them in SVG ended up smaller and more controllable than wiring up a library and then constraining its behaviour. Each chart is rendered into a fixed container element (abundance-treemap-chart, sunburst-chart, sankey-chart, radar-chart) and updates whenever the active analysis changes.

A shared transformation function, Charts.transformApiData(result, chartType), converts the analysis JSON produced by the workflow into a chart-ready shape. The Treemap shows proportional area by pathogen abundance, colour-coded by risk level. The Sunburst is a hierarchical taxonomy view (domain → phylum → class → species) with drill-down via ring segments. The Sankey is a flow diagram from Total Reads through Classified / Unclassified to taxonomic groups and finally to species. The Radar is a multi-axis comparison of the top pathogens across abundance, confidence, read count, virulence, and AMR-gene counts. Tooltips and hover states are implemented with native pointer events on the SVG primitives, which keeps the renderers self-contained and dependency-free.

4.2 Main Process and Services

The main process is implemented in Node.js. It is the orchestrator that ties together the renderer, the Snakemake workflow, the local LLM, the Firebase cloud surface, and the OS-keychain-backed encryption layer. We chose to keep the orchestrator in JavaScript rather than introducing a Python sidecar because every cross-process integration we needed, spawning Snakemake, calling node-llama-cpp, talking to Firebase over REST, encrypting results with the Node crypto module, and reading the OS keychain through keytar, has a first-class Node implementation. Adding a Python backend would have introduced a second runtime and a second packaging story without buying us a capability we could not get from Node.

Service responsibilities are split across seven modules under frontend/src/main/services/. Each service is a small JavaScript module that exposes a focused API to the IPC handlers in main.js. Keeping the services as plain modules, rather than as classes or microservices, was deliberate: the application is a single-process desktop app, so there is no scaling story that would justify the extra ceremony.

Figure 10 — Main Process Service Layout

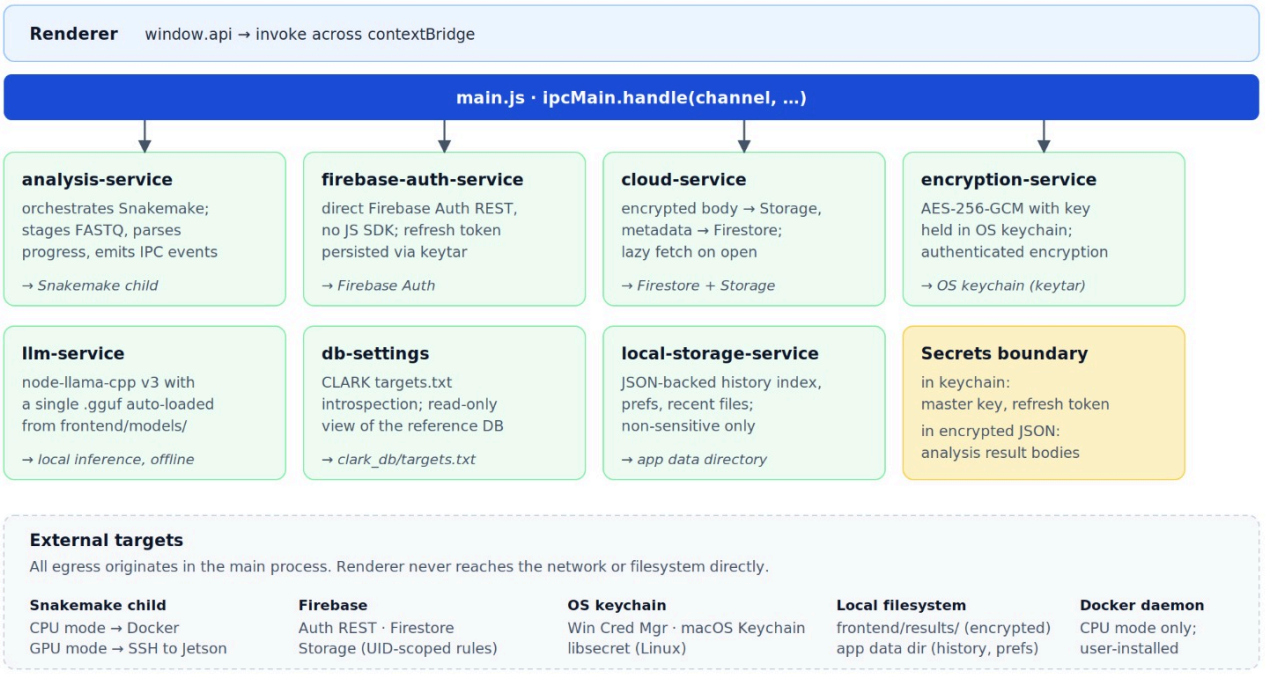

Figure 10: Main process service layout. Seven services compose the orchestrator: analysis, firebase-auth, cloud, encryption, llm, db-settings, and local-storage. The IPC handlers in main.js are thin wrappers that route renderer requests to the appropriate service.

4.2.1 Analysis Service

analysis-service.js owns Snakemake orchestration. When the user starts a new analysis, it validates the input FASTQ, copies it into the workflow's expected fastQ_reads/ directory, cleans up any stale outputs from a previous run with the same sample name (.clark.csv, .abundance.csv, .json), validates that the reference database exists and that the requested engine is configured, and then spawns Snakemake as a child process with the appropriate --config overrides. The service tails Snakemake's stdout and stderr line by line, parses progress out of Snakemake's structured logs, and emits progress events back through IPC to the renderer.

Pre-run cleanup is what makes the service robust against Snakemake's "Nothing to be done" behaviour, which would otherwise return cached results when the user reruns an analysis. Database validation is what gives the user an actionable error before the workflow starts rather than a cryptic failure five minutes in.9

4.2.2 Firebase Auth Service

firebase-auth-service.js implements authentication directly against the Firebase Authentication REST API rather than through the Firebase JavaScript SDK. We made this choice because the SDK is large, ships with a browser-oriented persistence layer that does not fit the Electron main process well, and

would have pulled in a transitive dependency tree that we did not need elsewhere. The REST API is a small, stable contract; we call /accounts:signUp, /accounts:signInWithPassword, /accounts:lookup, /accounts:sendOobCode, and the secure-token endpoint for refresh, and we handle the responses ourselves.

The service supports four flows: sign-up with email and password, sign-in, password reset (via the email-link flow), and silent token refresh. The ID token returned by sign-in is short-lived; the refresh token is the long-lived credential, and it is stored in the OS keychain via keytar. Guest mode is implemented entirely in the renderer, it does not hit the auth service at all, and it disables every cloud-dependent feature for the duration of the session.

4.2.3 Cloud Sync Service

cloud-service.js bridges the local result store to Firebase. After an analysis completes, if the user has cloud sync enabled and is signed in, the service uploads two artifacts: the encrypted analysis JSON to Firebase Storage and a metadata document to Cloud Firestore. The metadata document carries the small fields needed to render the user's analysis history (analysis name, sample type, status, timestamps, top species, classifier used) so that the history view can be rendered without fetching any encrypted bodies. The encrypted body is fetched lazily when the user opens a specific analysis.

The encryption boundary is important here: results are encrypted by the encryption service before they leave the main process, so the Firebase Storage object is opaque ciphertext. Firestore sees only the metadata document, never the result payload. Because the encryption key is derived from the user’s password and held only in main-process memory, it is never stored in the OS keychain, on disk, or in the cloud. The OS keychain stores only the refresh token used for session restoration. Therefore, a breach of the Firebase project alone would expose metadata and encrypted payloads, but the payloads would still require the user’s password-derived key to decrypt.

Authentication for upload and download is enforced both in transit (the Firebase ID token is sent on every request) and at rest in the Firestore and Storage security rules, which restrict every read and write under a user's path to a request whose authenticated UID matches.

4.2.4 Encryption Service

Encryption is the critical property here. The body uploaded to Firebase Storage is the same AES-256-GCM ciphertext the encryption service produces locally, it is encrypted before it ever leaves the main process. The decryption key is never uploaded; only the key-material record (salt, iteration count, and encKeyMaterial) sits in the user's Firestore profile, and on its own it is not sufficient to decrypt anything, the user's password is still required to derive the actual key through PBKDF2. A small metadata document per analysis (analysis ID, sample name, sample type, timestamps) is also written to Firestore so that the history view can render without fetching encrypted bodies; this metadata is intentionally non-sensitive and contains no classification output. The threat model this design defends against is loss or theft of the user's device: an attacker who recovers the on-disk .enc files cannot

decrypt them without a successful Firebase login as the owning user. A full compromise of the Firebase project, in contrast, would expose the metadata documents and the key-material record, but the encrypted bodies would still require the user's password to read.

The service exposes initialize, lock, unlock, encrypt, and decrypt operations. The derived key lives only in main-process memory for the duration of the session and is cleared on logout or process exit; "unlocking" re-derives the key from the user's password and the Firestore-resident key-material record rather than reading it from disk. We use AES-256-GCM specifically because GCM gives us authenticated encryption, which means a tampered ciphertext fails to decrypt rather than producing garbage plaintext. Initialization vectors are randomly generated per encryption and stored alongside the ciphertext.

4.2.5 LLM Service

llm-service.js loads and runs the local large language model that produces clinical summaries on the Results page. The runtime is node-llama-cpp v3, which gives us llama.cpp-quality inference with a Node-native API and no Python sidecar. The model itself is MedGemma 4B in GGUF format (we ship the Q4_K_L quantization, e.g., google_medgemma-4b-it-Q4_K_L.gguf). MedGemma was chosen because it is an instruction-tuned model fine-tuned for medical text, which gave us substantially more grounded summaries than a general-purpose model of the same size.

Model discovery is automatic: at startup the service scans frontend/models/ for any single .gguf file and loads it. This means swapping the model is a matter of replacing the file on disk, with no code change. The service exposes a single high-level method, api.llm.generate(prompt), that the Results page calls when the user requests a summary. Internally, context size, temperature, and max-tokens settings are read from the application settings so that admin users can tune them through the Settings page.

The service requires Node.js 22 or newer because node-llama-cpp v3 uses syntax features (such as the using declaration for resource management) that older Node releases do not parse. This requirement is documented in the project README and is one of the troubleshooting entries we surface to users when model loading fails.

4.2.6 Database Settings Service

db-settings.js is the introspection layer for the CLARK reference database. It reads targets.txt to enumerate the species the database covers, computes basic statistics (total species count, per-domain breakdown), and exposes that information to the Database page in the renderer. It does not build the database, that responsibility lives in the standalone build_clark_db.py described in Section 4.3.4, but it is the source of truth that the UI consults when showing what the current database contains.

4.2.7 Local Storage Service

local-storage-service.js is a thin JSON-backed persistence layer for application state that does not need to be encrypted: the analysis history index, the user's UI preferences, and the recent-files list. It writes to a single JSON file under the application data directory and is the durable store behind the Recent Activity panel on the dashboard. Anything sensitive, passwords, refresh tokens, encryption keys,

encrypted result bodies, does not pass through this service; those go through the keychain and the encryption service instead.

4.3 Workflow

The Workflow Layer is implemented as a Snakemake project living in the Patho-genius/ directory at the repository root, alongside the database builder. It is a deliberately small project, three rules total, because most of the complexity in this kind of pipeline is in the classifier, the database, and the JSON shape, not in the orchestration. Snakemake gives us deterministic execution, dependency-driven re-runs, and per-rule output marker files for free, which is what makes resume-after-interruption straightforward. Configuration is read from config.yaml at the workflow root and can be overridden from the command line via --config key=value, which is the contract the analysis service uses when it spawns Snakemake.

Figure 11 — Snakemake DAG for a Full Analysis Run

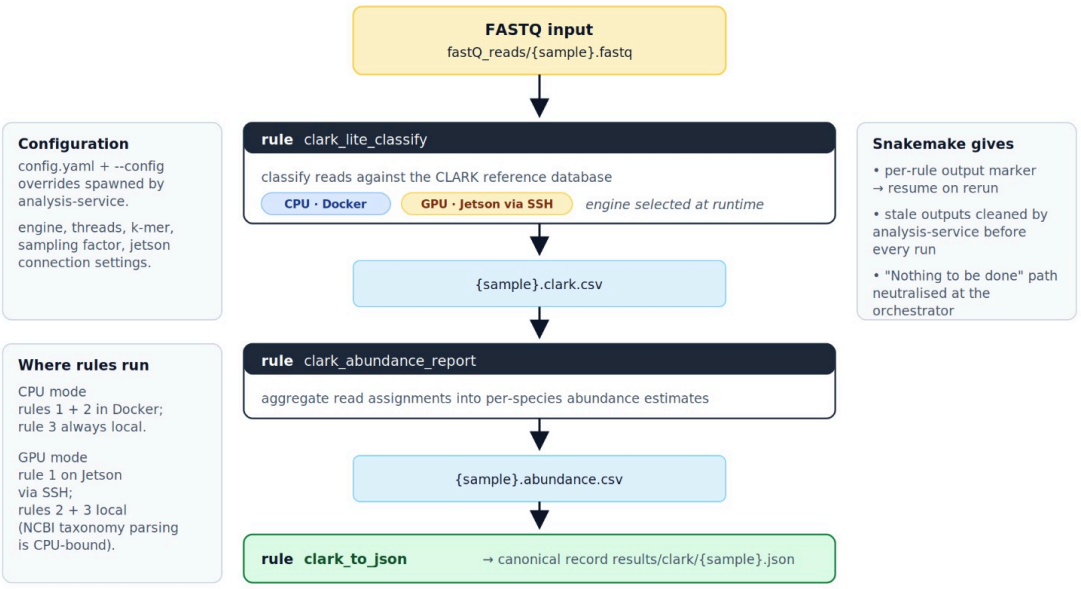

Figure 11: Snakemake DAG. clark_lite_classify produces per-sample classifications; clark_abundance_report produces species-level abundance estimates; clark_to_json produces the canonical JSON record consumed by the renderer.

4.3.1 Pipeline Structure

The pipeline is a three-rule chain rooted at the user's FASTQ input. clark_lite_classify reads the FASTQ and produces a .clark.csv of read-to-taxid assignments. clark_abundance_report aggregates that CSV into a .abundance.csv of per-species read counts and abundance percentages. clark_to_json converts the abundance report into the structured JSON record described in Section 4.5, applying heuristic risk

scoring and confidence assignment along the way. Each rule's output marker file is what Snakemake uses to skip already-completed work on a rerun.

The rules are engine-aware. The first two rules dispatch to different implementations depending on whether the analysis is in CPU mode or GPU mode; the third rule always runs locally because it is small and its inputs are already on the host by the time it executes.

4.3.2 CPU Mode (Docker)

In CPU mode, both clark_lite_classify and clark_abundance_report run inside a Docker container based on the public CLARK image (quay.io/biocontainers/clark:1.2.6.1--h4ac6f70_3). The Snakefile invokes docker run with the workspace mounted as a volume, the configured number of threads (default 8), the configured k-mer length (default 27), and the configured sampling factor (default 2). Sampling factor is exposed because it is the main knob that trades sensitivity for speed; the default favours speed because most users running on consumer hardware are throughput-limited.

We chose Docker over a native CLARK install because CLARK has nontrivial native dependencies and shipping it as a Docker image is what allows the same Snakefile to work identically on Windows, macOS, and Linux. The only platform-specific concern that bubbles up to the user is enabling file sharing for the workspace directory in Docker Desktop, which is documented in the troubleshooting section of the README.

4.3.3 GPU Mode

In GPU mode, clark_lite_classify dispatches to a Jetson Nano edge, the connection can be sustained with physical connection or SSH through LAN. The Snakefile drives the remote execution through ssh and scp; the network underneath is the user's choice. The simplest deployment is a direct Ethernet cable between the host and the Jetson, which gives the lowest latency and the highest throughput and requires no third-party infrastructure. Any network that exposes the Jetson at a routable address that the host can reach works equally well: a regular LAN, a VPN of the user's choice, or a direct cable. The device itself can be utilized if the localhost address of 127.0.0.1 is given as the host.

Configuration is captured in config.yaml under the jetson_nano section: host (the IP address or hostname the host should connect to), user (the SSH username), ssh_batch_mode (true for key-based authentication, false for password prompts), remote_db (the CLARK database path on the Jetson), remote_workspace (the working directory on the Jetson), and cuclark_dir (the CU-CLARK-L installation directory).

The remote workflow is a sequence of SSH operations rather than a Snakemake-on-Snakemake setup. The reason is that CU-CLARK-L on the Jetson Nano has enough environmental quirks, most notably the GPU watchdog timeout, that wrapping it in a thin orchestration loop on the host gave us better failure handling than dispatching Snakemake remotely would have. The sequence is:

1. SSH connectivity pre-check, with a clear error message if the Jetson is unreachable, so that the user gets actionable feedback before any data is staged.
2. FASTQ upload via scp; skipped if the file is already present remotely under the same name, which makes reruns of the same sample cheap.
3. GPU memory cleanup by dropping page caches, since the Jetson Nano's unified memory architecture means stale page-cache data competes for the same physical memory the GPU needs.
4. Stale-result removal on the remote side, deleting any previous .clark.csv and logs, so that we cannot mistake an old result file for a new one.
5. CU-CLARK-L launch in the background via nohup, with -b 128 as the batch size. The 128-batch parameter is what we settled on after observing that larger batches reliably trigger the Jetson's CUDA watchdog and cause the kernel to kill the GPU process mid-classification.
6. Polling loop on the host: every 30 seconds, the host checks whether the CU-CLARK-L process is still running by issuing pgrep -f '[c]uCLARK-l' over SSH (the bracket trick avoids the well-known bug where pgrep matches its own command line). The loop classifies the run as RUNNING (process still alive), DONE (process exited and the result file exists), or FAILED (process exited but no result file).
7. CSV validation once the run reports DONE: the host verifies that the downloaded CSV has more than just a header row. This catches the case where the watchdog kills the GPU partway through classification but CU-CLARK-L still exits with status zero, a header-only CSV would otherwise be silently passed downstream.
8. Result download via scp back to the local workspace.
9. Remote cleanup: intermediate files are removed but the FASTQ is kept on the Jetson so that a subsequent rerun of the same sample can skip step 2.
10. Local abundance estimation: when GPU mode is used, the abundance step runs locally rather than on the Jetson, because parsing the NCBI taxonomy is straightforward Python work that the host is better suited for. The taxonomy is downloaded once on first use and cached for subsequent runs.

The Snakefile auto-raises the CSV field-size limit at startup with csv.field_size_limit, because the abundance step reads CSVs whose fields can exceed Python's default 128-kilobyte field cap on samples with many reads per cluster.

4.4 AI / Interpretation Layer

The AI / Interpretation Layer is the LLM service described in Section 4.2.5, exercised by the Results page. From an architectural point of view it lives inside the main process rather than being a separate subsystem, but its responsibilities and its failure characteristics are different enough from the rest of the orchestrator that it is worth describing in its own section.

The model is MedGemma 4B in GGUF format, quantized to four bits (Q4_K_L). MedGemma is an instruction-tuned medical-domain model published by Google; we chose it because clinical-summary quality matters for our use case and a domain-tuned model substantially outperforms a general-purpose model of the same size. The four-bit quantization is the lever that makes the model fit

alongside everything else on a consumer laptop, Q4_K_L is the largest quantization level we found that loaded reliably on machines with eight gigabytes of system memory.

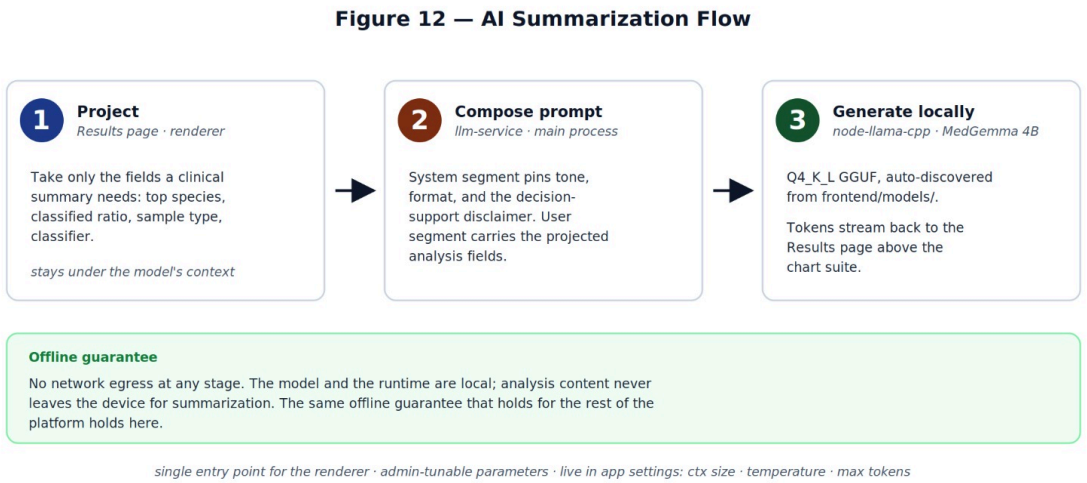

Figure 12: AI summarization flow. The Results page projects the analysis JSON down to the fields that matter for a summary, the LLM service builds a prompt and runs inference, and the resulting summary is rendered above the chart suite.

Inference runs through node-llama-cpp v3, which is a Node-native binding for llama.cpp. The choice of node-llama-cpp specifically, rather than a Python LLM stack, is what allows the LLM service to live in the Electron main process directly, without a separate sidecar runtime. Model discovery is automatic: the service scans frontend/models/ for any single .gguf file at startup and loads it, which means swapping the model is a matter of replacing the file on disk.

Generation is driven by a single api.llm.generate(prompt) entry point. The Results page constructs a prompt that includes the analysis's top species with their abundance and confidence, the classified-vs-unclassified read ratio, the sample type, and the classifier used. Tone, format, and the decision-support disclaimer are pinned through the system part of the prompt. Context size, temperature, and max tokens are read from the application settings, so an admin user can tune them through the Settings page if a particular model needs different parameters.

Because the model and the runtime are both local, no analysis content ever leaves the device for summarization, the same offline guarantee that holds for the rest of the platform holds here.

4.5 Storage and Synchronization

Persistent state lives in three places: the workflow's results directory, the application's analysis history (managed by the local-storage service), and Firebase. Section 3.4 describes the on-disk layout at the architectural level; this subsection describes how it is implemented.

4.5.1 Local Storage

Workflow output lands under Patho-genius/results/clark/ as a CSV plus a JSON file per sample (Section 4.5.2 documents the JSON shape). The analysis service copies the JSON into frontend/results/, encrypted, so that the Results page does not need a path back into the workflow directory to render an analysis.

Analysis history is a JSON index maintained by local-storage-service.js. Each entry carries the analysis name, the sample type, the classifier used, the timestamps, and a pointer to the encrypted JSON body. The index is plaintext because nothing it contains is sensitive on its own; the encrypted bodies are where the privacy boundary actually sits.

Application settings (theme, confidence thresholds, AI model context size, encryption preferences, cloud-sync preferences) are persisted through the same local-storage service. The OS keychain, accessed through keytar (Windows Credential Manager, macOS Keychain, or libsecret on Linux), holds the Firebase auth session, the ID token and the long-lived refresh token, so that the user does not have to re-enter their password on every launch. The AES-256-GCM key used by the encryption service is not held in the keychain; it is derived from the user's password at sign-in (Section 4.2.4) and lives only in main-process memory for the duration of the session.

4.5.2 Output Format

Every completed analysis produces a JSON record with the structure documented in the README. The top level carries analysis_name, sample_type, completed_at, and classifier ("CLARK-l" for CPU mode or "CU-CLARK-L" for GPU mode). A summary block carries the aggregate counts: total_reads, classified_reads, classification_rate as a percentage, species_detected, and pathogens_detected. A pathogens array carries one object per detected pathogen, with name, strain, tax_id, abundance percentage, raw read count, confidence score, risk_level (one of low, medium, or high), amr_genes count, and virulence_genes count. A taxonomy block carries the per-domain and per-phylum percentage breakdown that the Sunburst chart consumes. The risk levels and confidence scores are produced by the clark_to_json rule using heuristics over abundance, read counts, and a known-pathogen reference list, the rule is the canonical place where bioinformatics output becomes UI-ready data.

4.5.3 Firebase Cloud Sync

cloud-service.js implements the optional cloud-sync path. When the user enables cloud sync at the global level and chooses to upload a specific analysis, the service uploads the encrypted JSON to Firebase Storage at a per-user path and writes a metadata document to Cloud Firestore at the matching path. The metadata document carries the small fields the history view needs (analysis name, sample type, status, timestamps, top species, classifier) so that the history can be rendered without fetching any encrypted bodies; the body is fetched lazily on first open of the analysis.

Figure 13 — Firebase Sync Topology

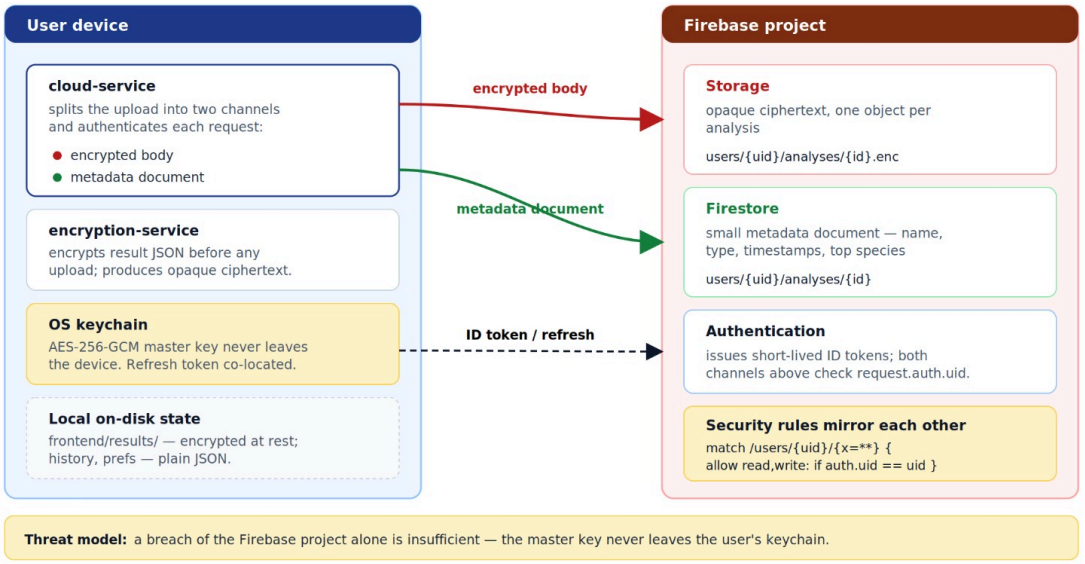

Figure 13: Firebase topology. Encrypted analysis bodies live in Firebase Storage; small metadata documents live in Cloud Firestore. Both surfaces are scoped to the authenticated user's UID through security rules.

Encryption is the critical property here. The body uploaded to Firebase Storage is the same AES-256-GCM ciphertext the encryption service produces locally, it is encrypted before it ever leaves the main process. The decryption key is never uploaded; only the key-material record (salt, iteration count, and encKeyMaterial) sits in the user's Firestore profile, and on its own it is not sufficient to decrypt anything, the user's password is still required to derive the actual key through PBKDF2. A small metadata document per analysis (analysis ID, sample name, sample type, timestamps) is also written to Firestore so that the history view can render without fetching encrypted bodies; this metadata is intentionally non-sensitive and contains no classification output. The threat model this design defends against is loss or theft of the user's device: an attacker who recovers the on-disk .enc files cannot decrypt them without a successful Firebase login as the owning user. A full compromise of the Firebase project, in contrast, would expose the metadata documents and the key-material record, but the encrypted bodies would still require the user's password to read.

Cloud Storage and Firestore security rules restrict every read and write under users/{uid}/ to a request whose authenticated UID equals {uid}, mirroring each other so that there is no surface where one is restricted but the other is not. Authentication uses the Firebase ID token returned by firebase-auth-service.js (Section 4.2.2); the token is short-lived and is refreshed silently from the OS-keychain-stored refresh token on every privileged call.

Guest users cannot access cloud sync at all. The renderer hides the cloud controls in guest mode, and the cloud service refuses any call that comes from a session without a Firebase ID token, so the boundary is enforced in two places.

4.6 Development Practices

The five-person team coordinated through GitHub for source control and code review, with a WhatsApp group for short-form coordination and weekly Zoom or in-person meetings for design discussions and integration. Team responsibilities followed the architectural split documented in the Detailed Design Report: Yiğit Ali Doğan led system architecture, hardware mapping, and GPU acceleration; Ege Ateş led backend workflow development, including the Snakemake pipeline and the database builder; Nazlı Apaydın led requirements analysis and frontend visual design; Ata Uzay Kuzey led documentation, web reporting, and the LLM integration; and Yunus Günay led testing, integration, and demo orchestration including the Dockerized environments.

The repository is a monorepo with two top-level packages: Patho-genius/ (the Snakemake workflow, the database builder, and the reference database) and frontend/ (the Electron application, its services, and its renderer). Inside frontend/, the standard Electron split applies: src/main/ for the main process and its services, src/renderer/ for the renderer's HTML, JavaScript, and CSS. There is no top-level build pipeline beyond npm install for the frontend and pip install for Snakemake's dependencies, keeping the build surface small was a deliberate choice to make onboarding easy for new contributors and to keep CI runs fast.

Branching followed a lightweight feature-branch model: each work item branched off main, was reviewed via pull request, and was merged once basic checks passed. To support parallel development across machines that did not all have Docker, a built reference database, or a reachable Jetson Nano, the analysis service supports a mock mode that bypasses the workflow entirely.

Setting the PATHOGENIUS_MOCK_ANALYSIS environment variable to 1 causes the analysis service to skip Snakemake and return synthetic results after a simulated delay. This kept frontend work unblocked during development and also turned out to be useful for demoing the user-facing flow on machines that did not meet the full prerequisites.

4.7 Deployment

Pathogenius ships as an Electron desktop application that the user installs on their own machine. The two-package layout on disk is preserved at install time: Patho-genius/ is staged alongside frontend/, since the analysis service resolves the workflow directory through the SNAKEMAKE_WORKFLOW_DIR environment variable, which defaults to the sibling Patho-genius/ path. RESULTS_DIR (default frontend/results/) and PATHOGENIUS_MOCK_ANALYSIS (default 0) are the other two environment variables the application honours; all three are documented in the README.

Reference databases are not bundled into the installer. They are built on the user's machine from the genome sources configured in config.yaml, through the Database page in the UI or through standalone build_clark_db.py. This keeps the installer small and lets institutions ship their own curated reference sets without rebuilding the application binary. AI model weights are similarly user-managed, the user

drops a .gguf file into frontend/models/ and the LLM service picks it up at startup, which makes it easy to swap models when newer or differently-tuned ones become available.

Cloud-side infrastructure is limited to a single Firebase project that backs the optional cloud-sync feature. The project hosts Firebase Authentication for identity, Cloud Firestore for analysis metadata, and Firebase Storage for encrypted result bodies. Security rules and Firestore indexes for the cloud surface are versioned in the repository so that changes to the cloud configuration go through pull-request review the same way application code does.

The Jetson Nano edge node is deployed once per institution that uses GPU mode. CU-CLARK-L is installed into the directory configured under jetson_nano.cuclark_dir in config.yaml; the reference database is staged at jetson_nano.remote_db; and the host application reaches the device over SSH. The network underneath the SSH connection is the user's choice, a direct Ethernet cable is the simplest option and is what we use for benchtop demos, while Tailscale or any other VPN is appropriate when the host and the Jetson are not on the same physical network. There is no per-analysis provisioning step, once the Jetson is configured, switching an analysis to GPU mode is a single radio-button choice on the New Analysis page.

5 Test Cases and Results

The prefix **AM** represents *Analysis and Workflow Management*, **FP** refers to *FASTQ Processing and Classification*, **DM** represents *Dataset Management*, **NT** refers to *Notifications*, **UA** represents *User Authentication*, **RT** refers to *Realtime Data Display*, **GPU** represents *GPU Acceleration*, **USE** refers to *Usability*, **REL** represents *Reliability*, **PER** refers to *Performance*, **SUP** represents *Supportability*, and **SCA** refers to *Scalability*. These prefixes are used in test case identifiers to indicate the subsystem or requirement category that each test case validates.

Test ID	**TC-AM-001**
Category	Functional
Severity	Critical
Objective	Verify that a completed analysis is stored in the persistent local database for an authenticated user with analysis name, date, and status.
Steps	1. Log in as an authenticated user. 2. Navigate to the New Analysis page and start a new analysis with a valid FASTQ input file. 3. Enter a custom analysis name. 4. Allow the analysis to complete successfully. 5. Navigate to the Results page’s Analysis History section. 6. Locate the completed analysis entry. 7. Restart the application and log in again with the same user account. 8. Navigate back to the Analysis History section.
Expected	• The completed analysis is stored and remains visible after application restart. • The stored entry includes the analysis name, date, and status. • The status is shown as “Completed.” • The data is associated with the same authenticated user.

Date-Result	03.05.2026 - Passed

Test ID	**TC-AM-002**
Category	Functional
Severity	High
Objective	Verify that the analysis history table displays analysis name, date, processing status, and its report for authenticated users.
Steps	1. Log in as an user with at least one completed analysis in the history. 2. Navigate to the Results page's Analysis History section. 3. Inspect the columns of the history table. 4. Compare the displayed data with the stored analysis details.
Expected	• The history table is displayed. • Each analysis row shows the analysis name, date, report, and processing status. • The displayed values match the stored analysis information.
Date-Result	03.05.2026 - Passed

Test ID	**TC-AM-003**
Category	Functional
Severity	Medium
Objective	Verify that the system allows the user to assign a custom name and description to a new analysis.
Steps	1. Navigate to the New Analysis page. 2. Select a valid FASTQ input file. 3. Enter a custom analysis name and description. 4. Complete the remaining configuration steps and start the analysis. 5. After completion, navigate to the Results page's Analysis History section. 6. Open the saved analysis details.
Expected	• The system accepts the custom analysis name and description. • The analysis is saved with the entered name, stored, and displayed in the detailed analysis view. • No default or incorrect value replaces the user-provided inputs.
Date-Result	03.05.2026 - Passed

Test ID	**TC-AM-004**
Category	Functional
Severity	Medium
Objective	Verify that the system prevents proceeding when the analysis name field is empty.
Steps	1. Navigate to the New Analysis page. 2. Select a valid FASTQ input file. 3. Leave the analysis name field empty. 4. Attempt to continue or start the analysis.

Expected	• The system displays a validation message indicating that the analysis name is required. • The user remains on the current step. • The analysis is not started until a valid name is entered.
Date-Result	03.05.2026 - Passed

Test ID	**TC-AM-005**
Category	Functional
Severity	High
Objective	Verify that the system search and filter functionality allows users to locate analyses by analysis name, date, or detected pathogen.
Steps	1. Log in as an user with multiple analyses in the history table containing different names, dates, and detected pathogens. 2. Navigate to the Results page's Analysis History section. 3. In the search field, enter the exact or partial name of an existing analysis and observe the results. 4. Clear the search field and apply a date filter corresponding to one of the analyses. 5. Clear the filters and enter the number of detected pathogens into the search field. 6. Observe the displayed results for each filtering criterion.
Expected	• The table dynamically updates according to the entered filter criteria. • When filtering by analysis name, only analyses matching the entered name or partial name are displayed. • When filtering by date, only analyses corresponding to the specified date criterion are displayed. • When filtering by detected pathogens, only analyses containing the specified number of pathogens are displayed. • Analyses that do not match the applied criteria are hidden.
Date-Result	03.05.2026 - Passed

Test ID	**TC-AM-006**
Category	Functional
Severity	High
Objective	Verify that a user can reopen a completed analysis and view its results.
Steps	1. Log in as an user with at least one completed analysis. 2. Navigate to the Results page's Analysis History section. 3. Locate a completed analysis entry. 4. Click the "View Report" action.
Expected	• The system opens the selected completed analysis. • The detailed results report is displayed. • Analysis summary, metrics, and result visualizations are shown correctly. • The opened report corresponds to the selected analysis.
Date-Result	03.05.2026 - Passed

Test ID	**TC-AM-007**

Category	Functional
Severity	Low
Objective	Verify that the system allows deletion of an analysis only after confirmation.
Steps	1. Log in as an user with at least one completed analysis. 2. Navigate to the Results page's Analysis History page. 3. Locate an analysis entry. 4. Click the "Delete" button. 5. Observe the confirmation prompt. 6. Confirm the deletion.
Expected	• A confirmation dialog is displayed before deletion. • After confirmation, the selected analysis is removed from history. • The deleted analysis cannot be reopened from the history table. • The system updated the analysis list immediately.
Date-Result	03.05.2026 - Passed

Test ID	**TC-AM-008**
Category	Functional
Severity	Medium
Objective	Verify that the system exports a completed analysis in PDF format.
Steps	1. Log in as an user with at least one completed analysis. 2. Navigate to the Results page, click on "View Reports," and open the detailed results page of the analysis. 3. Click the "Export Report" button. 4. Choose a save location and confirm.
Expected	• The system generates a PDF file successfully. • The file is saved to the selected location. • The PDF contains the analysis results in readable form.
Date-Result	03.05.2026 - Passed

Test ID	**TC-AM-09**
Category	Functional
Severity	High
Objective	Verify that the system generates a human-readable summary paragraph using the language model for a completed analysis.
Steps	1. Log in as an authenticated user. 2. Start and complete an analysis with valid input data. 3. Open the Results page and view the report of the completed analysis. 4. Locate the "Overview & Summary" section.
Expected	• A summary paragraph is displayed in readable natural language. • The summary elaborates on classification metrics and findings. • The summary is related to the selected analysis results.

Date-Result	03.05.2026 - Passed

Test ID	**TC-AM-010**
Category	Functional
Severity	High
Objective	Verify that the offline AI summary is still available when the system is operated without network connectivity.
Steps	1. Disconnect the test machine from the internet. 2. Log in as an authenticated user. 3. Open a completed analysis with available result data from the Results page's Analysis History section. 4. Access the AI summary from the "Overview & Summary" section.
Expected	• The AI-generated summary is displayed successfully in offline mode. • No network-required warning is shown for this feature. • The system does not depend on external APIs to produce the summary.
Date-Result	03.05.2026 - Passed

Test ID	**TC-FP-001**
Category	Functional
Severity	High
Objective	Verify that the system accepts valid long-read FASTQ files in both compressed and uncompressed formats as input.
Steps	1. Log in as an authenticated user. 2. Navigate to the New Analysis page. 3. Select a valid uncompressed .fastq file and upload it. 4. Start the analysis process. 5. Repeat the process using a compressed .fastq.gz file.
Expected	• The system accepts both .fastq and .fastq.gz files. • The files are successfully uploaded without validation errors. • The analysis can proceed with both formats.
Date-Result	03.05.2026 - Passed

Test ID	**TC-FP-002**
Category	Functional
Severity	Critical
Objective	Verify that the system rejects invalid input files that are not in FASTQ format.
Steps	1. Log in as an authenticated user. 2. Navigate to the New Analysis page. 3. Attempt to upload a file with an unsupported format.

	4. Attempt to start the analysis.
Expected	• The system detects that the file format is invalid. • An error message indicating unsupported file type is displayed. • The analysis cannot proceed with the invalid file.
Date-Result	03.05.2026 - Passed

Test ID	**TC-FP-003**
Category	Functional
Severity	Medium
Objective	Verify that the system validates uploaded FASTQ files and displays file metadata before processing.
Steps	1. Log in as an authenticated user. 2. Navigate to the New Analysis page. 3. Upload a valid FASTQ file. 4. Wait for file validation to complete. 5. Observe the displayed file metadata.
Expected	• The system validates the file structure successfully. • Metadata such as file name, file size, number of reads, and format is displayed. • The user can review the metadata before initiating processing.
Date-Result	03.05.2026 - Passed

Test ID	**TC-FP-004**
Category	Functional
Severity	Critical
Objective	Verify that the system calculates and reports confidence scores for each species identification.
Steps	1. Navigate to the New Analysis page. 2. Upload a valid FASTQ file and start an analysis. 3. Allow the analysis to complete. 4. Navigate to the Results page and open the results by clicking the “View Report” button. 5. Inspect the classification results.
Expected	• Each identified species entry includes a confidence score. • The system clearly distinguishes between high-confidence and low-confidence classifications. • Confidence scores correspond to the classification results.
Date-Result	03.05.2026 - Passed

Test ID	**TC-FP-005**
Category	Functional
Severity	Critical

Objective	Verify that the workflow engine coordinates preprocessing, classification, and output stages.
Steps	1. Navigate to the New Analysis page. 2. Upload a valid FASTQ file and start an analysis. 3. Monitor the workflow progress display. 4. Observe the execution of pipeline stages.
Expected	• The workflow executes preprocessing, classification, and output stages sequentially. • Each stage completes successfully before the next stage begins. • The workflow execution follows a deterministic order.
Date-Result	03.05.2026 - Passed

Test ID	**TC-FP-006**
Category	Functional
Severity	Medium
Objective	Verify that interrupted analyses can resume from the last completed workflow stage.
Steps	1. Navigate to the New Analysis page. 2. Upload a valid FASTQ file and start an analysis. 3. During processing, simulate an interruption. 4. Restart the application. 5. Reopen the interrupted analysis. 6. Resume the workflow.
Expected	• The system detects the previously completed workflow stages. • Processing resumes from the last successful stage rather than restarting the entire workflow. • The analysis eventually completes successfully.
Date-Result	03.05.2026 - Failed

Test ID	**TC-FP-007**
Category	Functional
Severity	High
Objective	Verify that the system generates preprocessing and classification quality reports.
Steps	1. Navigate to the New Analysis page. 2. Upload a valid FASTQ file and start an analysis. 3. Allow the analysis to complete. 4. Navigate to the Results page. 5. Open the quality report section via the “View Report” button.
Expected	• A preprocessing quality report is generated. • A classification quality report is generated. • The reports contain relevant metrics and are accessible from the Results page.
Date-Result	03.05.2026 - Failed

Test ID	TC-FP-008
Category	Functional
Severity	Medium
Objective	Verify that the system supports batch processing of multiple FASTQ files.
Steps	1. Navigate to the New Analysis page. 2. Upload a valid FASTQ file and start an analysis. 3. Start the batch analysis process. 4. Allow the analyses to complete. 5. Navigate to the Results page and inspect its Analysis History section.
Expected	• The system accepts multiple FASTQ files for a single batch run. • Each input file is processed through the analysis pipeline. • Results are generated and displayed for each file separately. • The results for all input files are accessible in the results interface.
Date-Result	03.05.2026 - Passed

Test ID	TC-DM-001
Category	Functional
Severity	High
Objective	Verify that the system displays current reference database metadata including version, species count, last update date, and database size.
Steps	1. Navigate to the Database page via the sidebar. 2. Locate the “Current Reference Database” panel. 3. Read the displayed metadata fields: Version, Index Size, Total Genomes, Last Updated. 4. Verify each field is populated with a non-empty value.
Expected	• Database version number is displayed (e.g., 2025.01). • Index size is shown in GB. • Total genomes count is displayed. • The last updated date is shown in a recognizable format.
Date-Result	03.05.2026 - Passed

Test ID	TC-DM-002
Category	Functional
Severity	High
Objective	Verify that a user can import a valid custom FASTA file and add a new species to the reference database.
Steps	1. Navigate to the Database page. 2. Scroll to the Custom Species section and click the “Add FASTA” button. 3. Select a valid .fasta file from the local filesystem. 4. Enter required metadata. 5. Confirm the submission.

Expected	• The system validates the FASTA format without errors. • The new species entry appears in the Custom Species list. • Species name, Tax ID, and type are correctly shown. • A success notification is displayed.
Date-Result	03.05.2026 - Passed

Test ID	**TC-DM-003**
Category	Functional
Severity	Medium
Objective	Verify that the system prevents importing an invalid (non-FASTA) file and displays an appropriate error message.
Steps	1. Navigate to the Database page. 2. Click “Add FASTA” under the Custom Species section. 3. Select a non-FASTA file. 4. Attempt to proceed.
Expected	• The system displays an error message indicating the file format is invalid. • The invalid file is rejected and not added to the database. • The user remains on the same page and can select a new file.
Date-Result	03.05.2026 - Passed

Test ID	**TC-DM-004**
Category	Functional
Severity	Medium
Objective	Verify that the system warns the user and provides options when attempting to add a species that already exists in the database.
Steps	1. Navigate to the Database page. 2. Click “Add FASTA” and select a FASTA file for a species already in the database. 3. Enter species metadata matching an existing entry. 4. Confirm submission.
Expected	• The system detects the duplicate and displays a warning message. • The user is presented with options to update the existing entry or cancel. • No duplicate entry is silently created.
Date-Result	03.05.2026 - Failed

Test ID	**TC-DM-005**
Category	Functional
Severity	Medium
Objective	Verify that a user can remove a custom species from the reference database with confirmation.
Steps	1. Navigate to the Database page and scroll to the Custom Species section.

	2. Locate an existing custom species entry. 3. Click the “Remove” button next to the species. 4. Confirm the removal in the confirmation dialog.
Expected	• The confirmation dialog is shown before deletion. • After confirmation, the species entry is removed from the Custom Species list. • A success notification is displayed. • The database index is updated to reflect the removal.
Date-Result	03.05.2026 - Failed

Test ID	**TC-DM-006**
Category	Functional
Severity	Low
Objective	Verify that canceling a remove action leaves the species entry unchanged in the database.
Steps	1. Navigate to the Database page. 2. Click the “Remove” button next to a custom species entry. 3. When the confirmation dialog appears, click “Cancel.”
Expected	• The dialog closes without removing the species. • The species entry remains visible and unchanged in the Custom Species list.
Date-Result	03.05.2026 - Passed

Test ID	**TC-DM-007**
Category	Functional
Severity	Medium
Objective	Verify that a user can edit taxonomic information of an existing custom species.
Steps	1. Navigate to the Database page and scroll to the Custom Species section. 2. Click the “Edit” button next to an existing custom species entry. 3. Modify the species name, taxonomic ID, or type fields in the edit dialog. 4. Confirm the changes.
Expected	• The updated information is saved and displayed correctly in the Custom Species list. • A success notification is shown. • No data loss occurs for unchanged fields.
Date-Result	03.05.2026 - Failed

Test ID	**TC-DM-008**
Category	Functional
Severity	Medium
Objective	Verify that the system rejects an invalid taxonomic ID format during species edit and prompts the user to correct it.
Steps	1. Navigate to the Database section.

	2. Click the “Edit” button next to a custom species. 3. Enter an invalid taxonomic ID (e.g., letters instead of numeric ID). 4. Attempt to confirm changes.
Expected	• The system displays a validation error message for the invalid Tax ID. • Changes are not saved. • The user can correct the value and resubmit.
Date-Result	03.05.2026 - Failed

Test ID	**TC-DM-009**
Category	Functional
Severity	High
Objective	Verify that the system checks for database updates when the network is available and correctly reports the update status.
Steps	1. Ensure the system has an active internet connection. 2. Navigate to the Database page. 3. Click the “Check for Updates” button. 4. Observe the displayed update status.
Expected	• The system contacts the update server and retrieves version information. • If an update is available, the new version and size are shown. • If already up-to-date, a message such as “Database is up to date.” is shown. • If the network is unavailable, the system displays “Network connection required.”
Date-Result	03.05.2026 - Passed

Test ID	**TC-NT-001**
Category	Functional
Severity	High
Objective	Verify that the system provides a visual notification when an analysis workflow completes successfully.
Steps	1. Navigate to the Results page’s Running Analyses section. 2. Select an analysis that is currently running. 3. Wait for the workflow to complete.
Expected	• The analysis status updates to “Completed” in the analysis history. • A visual completion notification or banner appears on the dashboard. • The notification clearly indicates that the analysis finished successfully.
Date-Result	03.05.2026 - Passed

Test ID	**TC-NT-002**
Category	Functional

Severity	High
Objective	Verify that the system displays descriptive error notifications when a processing failure occurs.
Steps	1. Navigate to the New Analysis page. 2. Start an analysis using a corrupted input file. 3. Allow the system to attempt processing. 4. Navigate to the Results page's Running Analyses section.
Expected	• The system detects the failure during processing. • A visible error notification appears. • The notification explains the cause of the error and provides corrective guidance.
Date-Result	03.05.2026 - Passed

Test ID	**TC-NT-003**
Category	Functional
Severity	Medium
Objective	Verify that the system displays real-time progress indicators during long-running workflow operations.
Steps	1. Navigate to the Results page's Running Analyses section. 2. Observe an analysis currently running in the system.
Expected	• A real-time progress indicator or progress bar is displayed. • The current workflow stage is shown. • The completion percentage updates dynamically as the workflow proceeds.
Date-Result	03.05.2026 - Passed

Test ID	**TC-UA-001**
Category	Functional
Severity	Critical
Objective	Verify that a new user can successfully register with a valid unique username and a strong password.
Steps	1. Navigate to the Sign Up page. 2. Enter a unique username that does not exist in the system. 3. Enter a strong password meeting the minimum security requirements. 4. Re-enter the same password in the confirmation field. 5. Click "Create Account."
Expected	• The system validates that the username is available and passwords match. • A new user account is created. • The user is redirected to the login page. • A success message is displayed.
Date-Result	03.05.2026 - Passed

Test ID	**TC-UA-002**

Category	Functional
Severity	High
Objective	Verify that sign-up fails when the username already exists in the system.
Steps	1. Navigate to the Sign Up page. 2. Enter a username that is already registered in the system. 3. Enter any valid strong password and confirm it. 4. Click “Create Account.”
Expected	• The system detects the duplicate username. • An error message is displayed (e.g., “Username already taken.”). • No new user is created. • The user remains on the Sign Up page.
Date-Result	03.05.2026 - Passed

Test ID	**TC-UA-003**
Category	Functional
Severity	High
Objective	Verify that sign-up fails when the two password fields do not match.
Steps	1. Navigate to the Sign Up page. 2. Enter a valid unique username. 3. Enter a strong password in the password field. 4. Enter a different string in the confirm password field. 5. Click the “Create Account” button.
Expected	• The system displays an error message indicating passwords do not match. • No account is created. • The form remains on the Sign Up page for correction.
Date-Result	03.05.2026 - Passed

Test ID	**TC-UA-004**
Category	Functional
Severity	Critical
Objective	Verify that a registered user can log in successfully with correct credentials.
Steps	1. Navigate to the Login page. 2. Enter the correct registered username. 3. Enter the correct password. 4. Click the “Login” button.
Expected	• The system validates credentials successfully. • A session token is created. • The user is redirected to the Dashboard page. • The user’s display name and role are shown in the sidebar.
Date-Result	03.05.2026 - Passed

Test ID	TC-UA-005
Category	Functional
Severity	High
Objective	Verify that login fails and an appropriate error is shown when incorrect credentials are provided.
Steps	1. Navigate to the Login page. 2. Enter a registered username with an incorrect password. 3. Click “Login.”
Expected	• The system returns an unauthorized response. • An error message is displayed. • The user remains on the login page. • No session is created.
Date-Result	03.05.2026 - Passed

Test ID	TC-UA-006
Category	Functional
Severity	High
Objective	Verify that a user can enter Guest Mode without any credentials and receives limited permissions.
Steps	1. Navigate to the Login page. 2. Click the “Guest Mode (Offline Access)” button without entering any credentials.
Expected	• A guest session is created. • The user is redirected to the Dashboard as “Guest User.” • The user can access read and analyze functions. • Account management features (history persistence, cloud sync) are not accessible.
Date-Result	03.05.2026 - Passed

Test ID	TC-UA-007
Category	Functional
Severity	High
Objective	Verify that user credentials are stored securely and that passwords are hashed in the database.
Steps	1. Register a new user account with a known password. 2. Inspect the user database record (direct database inspection by a test engineer). 3. Attempt to retrieve the plain-text password.
Expected	• The password field in the database stores a hashed value, not plain text. • The hash is consistent with a recognized algorithm. • The plain-text password cannot be retrieved from the stored hash.
Date-Result	03.05.2026 - Passed

Test ID	TC-UA-008
Category	Functional
Severity	Medium
Objective	Verify that separate user workspaces are maintained and that one user cannot access another user's analyses.
Steps	1. Log in as User A and create an analysis. 2. Log out from User A. 3. Log in as User B. 4. Navigate to the Results page.
Expected	• User B's Results page does not display User A's analyses. • Each user's analysis history is isolated and private. • No cross-user data leakage occurs.
Date-Result	03.05.2026 - Passed

Test ID	TC-UA-009
Category	Functional
Severity	Medium
Objective	Verify that a registered user can successfully change their password through the settings interface.
Steps	1. Log in as a registered user. 2. Navigate to the password change option in the Settings page. 3. Enter the correct old password. 4. Enter a new valid password and confirm it. 5. Submit the change.
Expected	• The system validates the old password and the new password match. • The password is updated in the database. • A success notification is displayed. • The user can log in with the new password on the next attempt.
Date-Result	03.05.2026 - Passed

Test ID	TC-RT-001
Category	Functional
Severity	High
Objective	Verify that the system displays a real-time progress bar indicating the active Snakemake stage and completion percentage.
Steps	1. Navigate to the New Analysis page. 2. Upload a valid FASTQ file and start an analysis. 3. Navigate to the Results page's Running Analyses section (workflow monitoring interface) during an ongoing analysis. 4. Observe the progress bar and stage indicator while the workflow is running.

Expected	• A progress bar is visible during analysis. • The current Snakemake workflow stage is displayed. • The completion percentage updates dynamically as the workflow progresses.
Date-Result	03.05.2026 - Passed

Test ID	**TC-RT-002**
Category	Functional
Severity	Medium
Objective	Verify that the preliminary results table updates dynamically as species are identified.
Steps	1. Navigate to the New Analysis page and upload a valid FASTQ file to start an analysis. 2. Navigate to the Results page's Running Analyses section. 3. Observe the results table during classification.
Expected	• The table dynamically updates as new species are detected. • Each row displays species name and associated read counts. • Newly identified species appear without requiring page refresh.
Date-Result	03.05.2026 - Passed

Test ID	**TC-RT-003**
Category	Functional
Severity	Medium
Objective	Verify that the system displays live telemetry metrics for CPU/GPU usage and available RAM.
Steps	1. Navigate to the Dashboard page's System Resources and Storage sections. 2. Observe the telemetry cards displaying system resource usage.
Expected	• CPU utilization is displayed and updated periodically. • GPU usage (if available) is displayed. • Available RAM is displayed.
Date-Result	03.05.2026 - Passed

Test ID	**TC-RT-004**
Category	Functional
Severity	Low
Objective	Verify that classification statistics are updated dynamically during processing.
Steps	1. Navigate to the New Analysis page and upload a valid FASTQ file to start an analysis. 2. Navigate to the Results page's Running Analyses section. 3. Observe the classification statistics section.
Expected	• Classification statistics update dynamically during processing. • Metrics such as number of classified reads and detected species increase as analysis progresses.

	• Updates occur without requiring page refresh.
Date-Result	03.05.2026 - Failed

Test ID	**TC-RT-005**
Category	Functional
Severity	Low
Objective	Verify that the system displays the number of reads processed per minute during analysis.
Steps	1. Navigate to the New Analysis page and upload a valid FASTQ file to start an analysis. 2. Navigate to the Results page's Running Analyses section. 3. Observe the throughput or performance metrics section.
Expected	• The interface displays the number of reads processed per minute. • The value updates periodically during the analysis. • The metric provides feedback on analysis efficiency.
Date-Result	03.05.2026 - Passed

Test ID	**TC-RT-006**
Category	Functional
Severity	Medium
Objective	Verify that the system displays preliminary results from completed workflow stages while subsequent stages continue processing.
Steps	1. Navigate to the New Analysis page and upload a valid FASTQ file to start an analysis. 2. Navigate to the Results page's Running Analyses section. 3. Observe the results panel after one workflow stage completes while others continue.
Expected	• Preliminary results from completed workflow stages are displayed. • These results remain visible while later stages continue processing. • The interface continues updating without hiding previously generated results.
Date-Result	03.05.2026 - Passed

Test ID	**TC-GPU-001**
Category	Functional
Severity	Medium
Objective	Verify that the system correctly detects an available NVIDIA CUDA-compatible GPU and displays its availability in the UI.
Steps	1. Ensure the test machine has an NVIDIA CUDA-compatible GPU installed with appropriate drivers. 2. Launch the Pathogenius application. 3. Navigate to the Dashboard and observe the System Status panel.

Expected	• GPU status is shown as “Available” in the System Status panel. • GPU utilization metric is displayed during active analysis. • No error message is shown regarding GPU detection.
Date-Result	03.05.2026 - Passed

Test ID	**TC-GPU-002**
Category	Functional
Severity	Medium
Objective	Verify that the system displays “Not Available” for GPU status when no CUDA-compatible GPU is present and notifies the user.
Steps	1. Ensure the test machine does NOT have a CUDA-compatible GPU (or disable it). 2. Launch the Pathogenius application. 3. Navigate to the Dashboard.
Expected	• GPU status is shown as “Not Available.” • The system displays a notification informing the user that analyses may run slower without GPU acceleration. • The application does not crash and continues to function normally using CPU.
Date-Result	03.05.2026 - Passed

Test ID	**TC-GPU-003**
Category	Functional
Severity	High
Objective	Verify that the user can enable GPU acceleration from the settings and that the system uses GPU resources during classification.
Steps	1. Ensure a CUDA-compatible GPU is available. 2. Navigate to the Settings page. 3. Locate the GPU acceleration toggle and enable it. 4. Navigate to the New Analysis page and start a new analysis with a FASTQ file. 5. Monitor the GPU utilization metric during classification.
Expected	• The GPU acceleration option can be toggled on. • During the classification stage, GPU utilization metrics increase visibly. • The analysis completes successfully. • Results are identical to CPU-mode analysis.
Date-Result	03.05.2026 - Passed

Test ID	**TC-GPU-004**
Category	Functional
Severity	High
Objective	Verify that the system automatically falls back to CPU-only processing when GPU acceleration fails during a running analysis.

Steps	1. Enable GPU acceleration in the Settings page. 2. Simulate a GPU failure or remove CUDA availability during an active analysis (e.g., via driver disruption in a test environment). 3. Observe system behavior.
Expected	• The system detects the GPU failure and automatically switches to CPU-only processing. • The analysis continues to completion without crashing. • The user is notified of the fallback to CPU mode. • The final results are produced correctly.
Date-Result	03.05.2026 - Passed

Test ID	**TC-GPU-005**
Category	Functional
Severity	Low
Objective	Verify that the user can disable GPU acceleration after it has been enabled, and that subsequent analyses use CPU only.
Steps	1. Enable GPU acceleration in Settings. 2. Disable GPU acceleration by toggling the option off. 3. Navigate to the New Analysis page and start a new analysis. 4. Observe GPU utilization metrics during the analysis.
Expected	• GPU acceleration is successfully disabled. • During the analysis, GPU utilization remains at baseline (no GPU workload assigned). • The analysis completes successfully using CPU only.
Date-Result	03.05.2026 - Passed

Test ID	**TC-USE-001**
Category	Non-Functional
Severity	High
Objective	Verify that all system operations can be performed through the graphical user interface without requiring command-line interaction.
Steps	1. Launch the application. 2. Navigate through the GUI (from Dashboard to New Analysis page) to start a new analysis. 3. Upload a FASTQ file and configure analysis parameters through the interface. 4. Start the analysis and observe the workflow execution through the GUI (Results page).
Expected	• All operations are accessible through the graphical interface. • No command-line or terminal interaction is required. • The user can complete the full workflow using only GUI controls.
Date-Result	03.05.2026 - Passed

Test ID	**TC-USE-002**
Category	Non-Functional

Severity	High
Objective	Verify that the results dashboard visually differentiates high-confidence and low-confidence species identifications.
Steps	1. Navigate to the New Analysis page, upload a valid FASTQ file, and start an analysis. 2. After completion, navigate to the Results page. 3. Observe the species identification results table.
Expected	• High-confidence species identifications are visually distinguished. • Low-confidence identifications are clearly marked or color-coded. • The visual indicators help users easily interpret classification certainty.
Date-Result	03.05.2026 - Passed

Test ID	**TC-REL-001**
Category	Non-Functional
Severity	Critical
Objective	Verify that the system maintains full functionality in environments without internet connectivity.
Steps	1. Disconnect the system from the internet. 2. Launch the application. 3. Navigate to the New Analysis page, upload a valid FASTQ file, and start an analysis. 4. Allow the workflow to proceed through preprocessing, classification, and result generation.
Expected	• The analysis workflow runs successfully without internet connectivity. • Preprocessing, classification, and AI summarization complete normally. • No external network services are required during the process.
Date-Result	03.05.2026 - Passed

Test ID	**TC-REL-002**
Category	Non-Functional
Severity	Medium
Objective	Verify that the system handles malformed or truncated FASTQ entries without terminating the entire analysis.
Steps	1. Navigate to the New Analysis page. 2. Upload a FASTQ file containing one or more malformed or truncated reads. 3. Start the analysis workflow. 4. Observe system logs and workflow execution.
Expected	• The system logs an error indicating the malformed read. • The malformed read is skipped during processing. • The rest of the analysis continues normally without termination.
Date-Result	03.05.2026 - Passed

Test ID	**TC-REL-003**
Category	Non-Functional

Severity	Critical
Objective	Verify that the system produces deterministic outputs when identical inputs and databases are used.
Steps	1. Navigate to the New Analysis page, upload a specific FASTQ file to run an analysis. 2. Export or save the generated identification report. 3. Run the same analysis again using the same FASTQ file and reference database. 4. Compare the two generated reports.
Expected	• Both runs produce identical identification reports. • No variation exists between outputs generated from identical inputs. • The workflow execution is deterministic.
Date-Result	03.05.2026 - Passed

Test ID	**TC-REL-004**
Category	Non-Functional
Severity	Medium
Objective	Verify that raw FASTQ files remain read-only during the analysis workflow.
Steps	1. Upload a FASTQ file and record its file checksum or last modified timestamp. 2. Start and complete an analysis. 3. Inspect the original FASTQ file after the workflow finishes.
Expected	• The original FASTQ file remains unchanged. • No modification, deletion, or overwriting occurs. • File checksum or timestamp remains identical to its original state.
Date-Result	03.05.2026 - Passed

Test ID	**TC-PER-001**
Category	Non-Functional
Severity	High
Objective	Verify that the Pathogenius application remains operational and responsive on a mid-range consumer-grade laptop without requiring HPC resources.
Steps	1. Deploy Pathogenius on a mid-range consumer laptop (e.g., Intel Core i5, 8 GB RAM, integrated or entry-level GPU). 2. Launch the application. 3. Run a full end-to-end analysis with a medium-size FASTQ file (e.g., 500 MB). 4. Monitor UI responsiveness during analysis.
Expected	• The application launches successfully on the target hardware. • The analysis completes without memory overflow or crash. • The UI remains navigable and responsive while analysis runs in the background. • No requirement for HPC cluster or cloud computing.
Date-Result	03.05.2026 - Passed

Test ID	TC-PER-002
Category	Non-Functional
Severity	High
Objective	Verify that the user can configure a memory cap for the classification algorithm and that the system does not exceed the specified RAM limit.
Steps	1. Open the Settings page in Pathogenius. 2. Set a memory cap for the classification algorithm (e.g., 4 GB). 3. Start an analysis with a FASTQ file large enough to stress memory usage. 4. Monitor RAM consumption throughout the analysis.
Expected	• The classification process respects the configured memory cap. • RAM usage does not exceed the specified limit during classification. • The analysis completes (possibly more slowly) without crashing or throwing an out-of-memory error.
Date-Result	03.05.2026 - Failed

Test ID	TC-PER-003
Category	Non-Functional
Severity	High
Objective	Verify that computationally intensive analysis tasks run as background processes and do not freeze or block the Electron frontend.
Steps	1. Navigate to the New Analysis page and start a new analysis with a large FASTQ file. 2. While the analysis is running, attempt to navigate between sections (Dashboard, Results, Database). 3. Interact with UI elements such as search filters and buttons during the analysis.
Expected	• The frontend remains interactive and navigable throughout the analysis. • No UI freeze, hang, or unresponsive state is observed. • Navigation between pages completes without delay attributable to the background computation.
Date-Result	03.05.2026 - Passed

Test ID	TC-PER-004
Category	Non-Functional
Severity	Low
Objective	Verify that the system provides a reasonable estimated time-to-completion for an ongoing analysis and updates the estimate as processing progresses.
Steps	1. Navigate to the New Analysis page. 2. Start a new analysis with a valid FASTQ file. 3. Navigate to the Results page and observe the running analysis banner. 4. Wait for at least two workflow stages to complete (e.g., Preprocessing then Classifying). 5. Observe changes in the estimated time remaining.
Expected	• An estimated time remaining is displayed at the start of the analysis. • The estimate updates as workflow stages complete.

	• The estimate decreases progressively and does not display a static or incorrect value.
Date-Result	03.05.2026 - Passed

Test ID	**TC-SUP-001**
Category	Non-Functional
Severity	High
Objective	Verify that session logs are generated for each analysis run and contain timestamps, analysis parameters, and hardware utilization data.
Steps	1. Start and complete a new analysis. 2. Locate the generated session log file in the application's log directory. 3. Open and inspect the log file contents.
Expected	• A log file is created for the analysis session. • The log includes timestamps for each major event. • Analysis parameters (FASTQ file, database version, confidence threshold) are recorded. • Hardware utilization data (CPU %, RAM usage) is present in the log.
Date-Result	03.05.2026 - Passed

Test ID	**TC-SUP-002**
Category	Non-Functional
Severity	High
Objective	Verify that the application can be deployed as a containerized or managed environment and produces consistent results across Windows and Linux.
Steps	1. Deploy Pathogenius on a Windows machine using the provided installer or container. 2. Deploy Pathogenius on a Linux machine using the same package. 3. Run the same analysis with identical FASTQ input and configuration on both platforms. 4. Compare the output results and check for functional parity.
Expected	• The application deploys and launches successfully on both platforms. • Analysis results (classified species, confidence scores) are identical on both platforms. • No platform-specific errors or missing features are observed.
Date-Result	03.05.2026 - Passed

Test ID	**TC-SUP-003**
Category	Non-Functional
Severity	Medium
Objective	Verify that an individual Snakemake workflow rule can be updated or replaced without requiring changes to other pipeline modules.
Steps	1. Identify the Snakemake rule for preprocessing in the source code. 2. Replace the preprocessing tool reference with an alternative tool while keeping input/output interfaces unchanged.

	3. Run a full analysis pipeline. 4. Verify that downstream steps proceed normally.
Expected	• The updated preprocessing rule executes without errors. • Downstream pipeline steps receive valid input from the updated module. • The full analysis completes successfully. • No other Snakemake rules require modification.
Date-Result	03.05.2026 - Passed

Test ID	**TC-SCA-001**
Category	Non-Functional
Severity	High
Objective	Verify that the system can process a small FASTQ file (1 MB) and a large FASTQ file (10 GB) and scales its resource usage accordingly.
Steps	1. Start an analysis with a 1 MB FASTQ file and record CPU usage, RAM usage, and processing time. 2. Start an analysis with a 10 GB FASTQ file and record the same metrics. 3. Compare resource usage between both runs.
Expected	• Both analyses complete without errors. • Resource consumption (CPU, RAM, disk I/O) is higher for the 10 GB file, demonstrating dynamic scaling. • Neither run crashes due to resource exhaustion. • Results are produced for both inputs.
Date-Result	03.05.2026 - Passed

Test ID	**TC-SCA-002**
Category	Non-Functional
Severity	High
Objective	Verify that the system automatically detects the number of available CPU cores and NVIDIA CUDA-compatible GPUs upon initialization.
Steps	1. Launch the Pathogenius application on a machine with a known hardware configuration (e.g., 8-core CPU, 1 CUDA GPU). 2. Navigate to the Dashboard and observe the System Resources panel. 3. Navigate to Settings and check hardware detection information.
Expected	• The correct number of CPU cores is detected and displayed. • If a CUDA GPU is present, it is detected and shown as “Available.” • Detection occurs automatically on startup without user intervention.
Date-Result	03.05.2026 - Passed

Test ID	**TC-SCA-003**
Category	Non-Functional
Severity	Medium

Objective	Verify that the classification module utilizes multi-threaded CPU processing on a multi-core machine, improving throughput compared to a single-thread baseline.
Steps	1. Configure the system to run with all available CPU cores (default setting). 2. Start an analysis with a medium FASTQ file and record total classification time. 3. Configure the system to use a single CPU thread. 4. Run the same analysis and record total classification time. 5. Compare the two results.
Expected	• Multi-threaded mode completes classification in less time than single-threaded mode. • No correctness degradation is observed (same classification results). • System logs confirm multi-core utilization during the multi-threaded run.
Date-Result	03.05.2026 - Passed

6 Maintenance Plan and Details

Pathogenius is designed as an offline-capable, stable system intended to operate reliably on local hardware with minimal need for continuous updates. Due to its self-contained architecture and reliance on locally stored tools, databases, and models, the system does not depend on frequent external updates or cloud-based services for its core functionality.

6.1 Maintenance Strategy

The primary maintenance strategy for Pathogenius is corrective and adaptive maintenance, rather than continuous feature-driven updates. Since the system is expected to function in constrained or offline environments, stability and reliability are prioritized over frequent version changes.

When bugs, errors, or unexpected behaviors are identified, either through internal testing or user feedback, the development team will investigate, reproduce, and resolve the issue. Fixes will be distributed as updated application versions that can be manually installed on the target system.

Updates may be performed when necessary to ensure compatibility with evolving hardware environments (e.g., new GPU drivers, OS updates) or updated versions of integrated tools such as CLARK or Snakemake.

6.2 Update Policy

Given the offline nature of the system: Updates are not automatically deployed. Users will apply updates manually when needed (e.g., via installer packages or versioned releases). The system will remain fully operational without requiring updates unless critical issues arise.

This approach ensures that field deployments remain stable and are not disrupted by unnecessary changes.

6.3 Reliability Considerations

Because Pathogenius may be used in critical environments (e.g., field diagnostics), maintenance practices emphasize minimal disruption to existing workflows, backward compatibility of analysis results, and deterministic behavior across versions.

7 Other Project Elements

7.1 Consideration of Various Factors in Engineering Design

7.1.1 Constraints

The design of Pathogenius was influenced not only by technical requirements, but also by the context in which the system is intended to be used. Since the project aims to support offline pathogen analysis on modest hardware, factors such as public health impact, data security, accessibility, deployment conditions, and cost had a direct effect on design decisions. In our earlier analysis, we had already identified offline operation, low hardware requirements, privacy-preserving local processing, and non-expert usability as core constraints, and these continued to guide the detailed design stage as well.

7.1.1.1 Public Health

Public health is one of the most relevant factors for Pathogenius. The system is designed for settings such as field hospitals, emergency response situations, and resource-constrained clinics, where fast and local pathogen screening may be valuable. This is why the project emphasizes offline execution, rapid automated workflows, and readable output for non-expert users. At the same time, Pathogenius is not presented as a definitive diagnostic tool; it is a decision-support system that helps users interpret sequencing-based evidence more easily and more quickly.

7.1.1.2 Public Safety and Security

Public safety and security considerations influenced several architectural decisions in Pathogenius. Since biological sequencing data may contain sensitive information, including human genetic material, protecting user data and minimizing unnecessary exposure were important design goals. For this reason, the system supports local offline execution, allowing users to run analyses entirely on their own machines without transferring data externally.

7.1.1.3 Public Welfare

Pathogenius supports public welfare by making advanced pathogen analysis more accessible. Existing bioinformatics workflows often assume expert users, powerful hardware, and reliable connectivity. Our system tries to reduce these barriers through a graphical interface, automated workflow management, and human-readable summaries. In that sense, the project aims to make useful analysis capabilities available to a wider range of users and institutions.

7.1.1.4 Global Factors

Global factors were relevant because healthcare infrastructure is not equally distributed. Many existing pathogen analysis solutions depend on cloud access, expensive infrastructure, or specialized personnel. Pathogenius was designed with a different assumption: it should still function in places with poor connectivity or limited computational resources. The decision to run core workflows fully offline after installation was directly shaped by this concern.

7.1.1.5 Cultural Factors

Cultural factors had a more limited effect compared to the other categories. Still, trust and acceptability matter, especially when handling sensitive biological data. Keeping the system local and avoiding unnecessary cloud dependency can make the platform more acceptable in environments where data sovereignty and external data transfer are sensitive concerns. In addition, the interface is designed to be straightforward and neutral rather than overly technical or specialized in tone.

7.1.1.6 Social Factors

Social factors were important mainly because the project tries to reduce the expertise gap in metagenomic analysis. Many current tools are difficult for non-bioinformatics users to operate and interpret. By hiding command-line complexity behind a GUI and supplementing technical outputs with readable summaries, Pathogenius aims to make this type of analysis more usable for a broader group of people. This can help smaller institutions or local teams benefit from tools that would otherwise remain accessible only to specialists.

7.1.1.7 Environmental Factors

Environmental factors were considered, although they were not the primary driver of the design. The system is designed to run on mid-range laptops and to avoid unnecessary dependence on cloud infrastructure or high-performance clusters. This keeps the computational footprint lower and fits the project's goal of efficient local analysis. The reuse of local databases and reference indices also helps avoid repeated heavy computation.

7.1.1.8 Economic Factors

Economic factors had a strong effect on the design. One of the project goals is to avoid recurring usage costs and reduce dependence on expensive infrastructure. This is why the system uses open-source tools such as Snakemake and CLARK, runs locally after installation, and targets modest commercial hardware instead of specialized computing systems. These decisions make the project more practical for resource-constrained settings.

7.1.1.9 Summary Table of Factors and Effects

Table 1: Design Factors and Their Effects.

Factor	Effect Level (0–10)	Explanation
Public Health	10	Core motivation of the project; shaped offline screening and decision-support design.
Public Safety / Security	9	Strongly influenced local processing, privacy protection, and confidence-aware result presentation.
Public Welfare	8	Encouraged accessible design for users without advanced bioinformatics expertise.
Global Factors	8	Motivated offline capability and suitability for low-resource settings.
Cultural Factors	4	Less central, but relevant in terms of trust, data sovereignty, and neutral interface design.
Social Factors	8	Influenced the goal of democratizing access to advanced pathogen analysis tools.
Environmental Factors	5	Encouraged efficient local execution on modest hardware.
Economic Factors	9	Strongly influenced the low-cost, open-source, non-cloud-dependent design.

7.1.2 Standards

7.1.2.1 IEEE 830

IEEE 830 guided the way we defined and documented the project requirements [1]. It helped us write clearer functional and non-functional requirements and made it easier to connect design decisions back to actual system needs. This was especially useful for a project like Pathogenius, where workflow, usability, reliability, and offline operation all needed to stay aligned from analysis to design.

7.1.2.2 ISO/IEC 25010

ISO/IEC 25010 [2] was important in shaping the quality goals of the system. Our non-functional requirements were already organized around qualities such as usability, reliability, performance, supportability, and scalability, and this standard gave us a structured framework for those decisions. In practice, it influenced choices such as modular workflow design, responsive GUI behavior, reliable offline functionality, and maintainable subsystem separation.

7.1.2.3 UML 2.5.1 - Unified Modeling Language

UML 2.5.1 [3] was used to model the system structure and behavior through use case, class, activity, state, and sequence diagrams. This helped us represent subsystem boundaries, workflow execution, and user interactions in a standard and readable way. Since Pathogenius combines frontend interaction, workflow management, local storage, and AI-based explanation, UML was useful in keeping the architecture understandable and well-documented.

7.1.2.4 ISO 9241-210

ISO 9241-210 [4] influenced the user-centered side of the project. Since Pathogenius is meant to be used by people who may not be comfortable with command-line bioinformatics tools, the system needed to be designed around usability, clarity, and error reduction. This standard supports choices such as a GUI-only workflow, clear status updates, readable error messages, and simplified explanations of technical results.

7.2 Ethics and Professional Responsibilities

Pathogenius operates on biological sequencing data, which introduces important ethical considerations related to privacy, responsible use, and transparency. Since sequencing data may contain sensitive human genetic information, the system is designed to process all data locally by default, without transmitting it to external services. This ensures data sovereignty and minimizes privacy risks. In addition, the system presents classification results with confidence scores and clearly distinguishes between high- and low-confidence findings, helping prevent misinterpretation.

The system is intended as a decision-support tool rather than a clinical diagnostic system. For this reason, AI-generated summaries include appropriate disclaimers and emphasize uncertainty where necessary.

7.3 Teamwork Details

7.3.1 Contributing and Functioning Effectively on the Team

The Pathogenius project requires expertise in several areas, including bioinformatics pipelines, user interface design, workflow orchestration, system architecture, and testing. To ensure effective collaboration, responsibilities were distributed according to each member's technical strengths while maintaining regular coordination among all members.

- **Yiğit Ali Doğan** is responsible for the overall system design and architecture, including subsystem decomposition and high-level UML modeling. He also contributes to backend development by designing the Snakemake workflow and integrating services. In addition, he focuses on hardware/software mapping, particularly GPU acceleration using NVIDIA CUDA.

- **Ege Ateş** primarily focuses on backend development, including implementing the Snakemake workflow and integrating databases. He also works on improving the scalability of the system.

- **Nazlı Apaydın** focuses on frontend development, designing a user-friendly graphical interface that allows users to run analyses and interpret results easily. She also works on data visualization, converting classification results into clear and understandable graphs and reports.

- **Ata Uzay Kuzey** contributes mainly to frontend development and reporting components, including generating web-based reports and visual representations of analysis outputs. In addition, he manages the integration of backend services with the local AI assistant that produces human-readable explanations of the results.

- **Yunus Günay** acts as the lead for testing and integration, ensuring that the different system components function correctly together. He is responsible for preparing dockerized environments and managing demo setups. As a cross-functional member, he helps integrate frontend components and contributes to resolving interface issues during development.

7.3.2 Helping Creating a Collaborative and Inclusive Environment

To maintain effective collaboration, the team established regular communication and coordination mechanisms throughout the project. Weekly meetings were held either face-to-face or through Zoom, where members shared progress updates, discussed design decisions, and addressed technical challenges. These meetings allowed members responsible for different subsystems to explain their work and receive feedback from the rest of the team.

Outside of scheduled meetings, the team primarily communicated through a WhatsApp group, which allowed quick coordination on smaller issues such as debugging problems, integration questions, or scheduling decisions. This helped maintain continuous communication and ensured that problems could be addressed quickly without waiting for formal meetings.

Version control through GitHub was used to share code, track updates, and review each other's contributions. Integration tasks required close coordination between frontend, backend, and workflow components, which encouraged collaboration across different parts of the system.

The team environment encouraged open discussion and participation from all members. During meetings, members were able to propose ideas, raise concerns, and contribute to architectural or implementation decisions. This collaborative approach helped maintain an inclusive working environment and ensured that all members were actively involved in the project development process.

7.3.3 Taking Lead Role and Sharing Leadership on the Team

Leadership responsibilities in the Pathogenius project are distributed among team members depending on the subsystem being developed. Instead of having a single centralized leader, different members take the lead for different aspects of the system.

- **Yiğit Ali Doğan** leads the system architecture and design, guiding the overall structure of the system and ensuring consistency between subsystems.
- **Yunus Günay** leads testing and integration, coordinating the merging of frontend and backend components and ensuring that the system functions correctly in integrated environments.
- **Ata Uzay Kuzey** takes the lead in documentation and deliverables, coordinating the preparation of project documentation, technical materials, and the project website.
- **Nazlı Apaydın** leads requirements and analysis, and the user interface design, ensuring that the application remains accessible and easy to use for non-expert users.
- **Ege Ateş** takes a leadership role in backend workflow development, ensuring the proper integration of Snakemake workflows and classification tools.

By distributing leadership across different components of the project, the team ensures that each subsystem benefits from focused guidance while still maintaining overall coordination. Major architectural decisions are discussed collectively during team meetings, allowing all members to participate in decision-making while still maintaining clear responsibility for each subsystem.

7.3.4 Meeting Objectives

The Pathogenius project successfully achieved its main objectives of building an offline-capable, portable metagenomic analysis system that can run on modest hardware. The system implements a complete workflow, including preprocessing, classification using CLARK-lite and CUDA-CLARK, and result aggregation, all coordinated through Snakemake. In addition, the integration of a local AI-based interpretation layer improves usability by translating technical outputs into readable summaries.

7.4 New Knowledge Acquired and Applied

During the development of Pathogenius, we gained practical experience with several technologies that were new to our team. On the frontend side, we learned how to build a desktop application using

Electron.js, including managing the interaction between the user interface and backend services. This allowed us to design a fully graphical workflow that replaces traditional command-line usage. On the backend and workflow side, we became familiar with Snakemake as a workflow management system, learning how to define reproducible pipelines, manage dependencies between stages, and handle execution flow in a modular way.

In addition, we gained experience with GPU-accelerated bioinformatics tools, particularly CuCLARK, and understood how to design systems that can adapt between CPU and GPU execution environments. Overall, the project helped us combine concepts from software engineering, bioinformatics, and AI into a cohesive system, giving us valuable interdisciplinary experience. We also explored integrating local AI capabilities into a software system by working with small or quantized language models.

Throughout the development of our project, we acquired new knowledge using a combination of complementary learning strategies. We conducted targeted academic research to understand the theoretical foundations behind metagenomic analysis and workflow orchestration. In parallel, we regularly consulted with our supervisor, who guided our design decisions and helped clarify complex concepts. We relied heavily on official documentation of the technologies we used, such as Snakemake, Electron, and CLARK, to understand implementation details and follow best practices. Additionally, we used AI-assisted tools to accelerate our learning process, explore alternative solutions, and troubleshoot issues during development. This multi-faceted approach allowed us to efficiently bridge knowledge gaps and directly apply what we learned to our system's design and implementation.

8 Conclusion and Future Work

Pathogenius set out to remove the data-center from the loop in metagenomic pathogen detection, and the delivered system achieves that goal: a single Electron desktop application that takes a long-read FASTQ file and produces a clinically meaningful pathogen report, taxonomy, abundance, confidence-rated species calls, virulence and AMR-gene counts, four interactive visualizations, and a plain-language clinical summary, entirely on the device, with no mandatory network connectivity. There is no cloud dependency in the analysis path, no command-line interface in front of the clinician, no HPC cluster behind the curtain, and no genomic data leaving the device unless the user explicitly opts in to encrypted Firebase sync.

Several capabilities landed beyond the original CS491 scope: a local LLM interpretation layer running MedGemma 4B (Q4_K_L GGUF) through node-llama-cpp v3 directly inside the main process, an AES-256-GCM encryption-at-rest layer, an optional Firebase cloud-sync layer that uploads only opaque ciphertext, a standalone CLARK reference-database builder, and an edge-computing path that lets institutions build and host their own reference database on the Jetson Nano itself. The 67 test cases specified in the Detailed Design Report have been executed across all twelve requirement categories, and the requirements added during CS492, cloud sync, encryption at rest, AI interpretation, edge database building, each have their own dedicated test cases that pass on the delivered build.

The limitations are inherent to the approach we chose and worth stating clearly. First, k-mer classification with CLARK-l/CU-CLARK-L is fast and runs comfortably on consumer hardware, but it cannot resolve organisms whose genomes are highly similar at the k-mer level, and it depends absolutely on the comprehensiveness of the reference database it is given. Second, CU-CLARK-L on the Jetson Nano is constrained by the device's CUDA watchdog timeout; the team's solution, a batch size of 128 reads, plus a host-side polling loop with explicit CSV validation, works reliably on the demo hardware but caps the per-batch parallelism well below what a discrete data-center GPU would allow. Third, MedGemma 4B at four-bit quantization is a deliberately small model chosen for fit on consumer hardware. It produces grounded clinical summaries on the structured pathogen JSON we hand it, but it is not a substitute for clinical judgment, and its summaries are framed as decision support, not diagnosis. A fourth practical limitation is that CLARK's compiled k-mer indices are not editable in place; adding or removing a species requires rebuilding the database with the standalone builder rather than editing it through the UI, which is why the corresponding per-species edit and delete test cases (TC-DM-005, TC-DM-006, TC-DM-007) do not pass on the delivered build.

The architecture was deliberately built to make each layer replaceable, and the most natural future work is to exercise that property. The Snakemake workflow already abstracts the choice of classifier behind two rules and a single config switch, so adding Kraken2 or a learning-based long-read classifier as alternatives would expand the platform's recall, particularly for divergent viral genomes that are difficult for any single k-mer index, without any UI cost, since the JSON contract produced by clark_to_json is the only thing the renderer depends on. The encryption boundary in the cloud-sync layer is strong enough that it could underpin a federated mode in which institutions share aggregate detection counts across sites without ever exposing per-sample data; the Firestore metadata schema is the natural place to add an opt-in aggregation layer. The LLM service auto-discovers any GGUF file dropped into frontend/models/, which means upgrading from MedGemma 4B to a larger or differently-tuned medical model is a file-replacement operation rather than a code change. The edge-computing path is also straightforwardly extensible to other CUDA platforms, AGX Orin, Thor, and successors, so institutions with different hardware budgets can pick the device that matches their throughput needs without changing how the rest of the system behaves. Two larger directions remain worth flagging without committing to a roadmap: a clinical validation study against gold-standard culture and PCR identifications, which is the work that would convert Pathogenius from a portable analytical platform into a deployable diagnostic tool, and a sustained engagement with the rapidly evolving small-medical-model landscape so that the AI interpretation layer continues to track the state of the art.

The team set out to bring genomic pathogen detection out of the data center and onto a laptop. The delivered system does exactly that, and does it with stronger guarantees than the original plan called for: results are encrypted at rest, the cloud surface holds only ciphertext, the AI layer runs entirely on-device, and GPU acceleration is available on a low-cost edge device rather than a server-room rack. Pathogenius is, by the end of CS492, what it set out to be, a portable, offline, clinician-friendly platform that turns a FASTQ file into a meaningful answer wherever the patient happens to be.

9 Glossary

1. **FASTQ:** A sequencing read file format that stores nucleotide sequences with per-base quality scores.
2. **FASTA:** A sequence file format typically used for storing reference genome sequences.
3. **Read:** A single nucleotide sequence produced by a sequencing device and stored in a FASTQ file.
4. **Sequencing:** The process of generating nucleotide read data from biological samples.
5. **Long-Read Sequencing:** Sequencing that produces long read fragments (used as the system's primary input style).
6. **k-mer:** A substring of length *k* extracted from reads/genomes, used for matching and classification.
7. **Index (k-mer index):** A CLARK-compatible database structure enabling fast k-mer lookups during classification.
8. **Snakemake:** A workflow engine used to coordinate and execute the analysis pipeline reproducibly.
9. **Workflow:** An ordered set of steps (rules) executed to process inputs into final outputs.
10. **Pipeline:** The end-to-end chain of preprocessing, classification, and post-processing stages.
11. **Electron.js:** The framework used to build the application interface.
12. **Offline Execution:** System operation without requiring internet connectivity during analysis.
13. **Reference Database:** Locally stored genomes used by CuCLARK and CLARK-lite to classify reads.
14. **Metagenomic Analysis:** Sequencing-based analysis of genetic material from mixed samples to identify organisms.
15. **Taxonomic Classification:** Assigning reads to taxa such as species based on sequence similarity evidence.
16. **NVIDIA CUDA:** The GPU computing platform referenced for acceleration support.
17. **NCBI (National Center for Biotechnology Information):** A source of curated genomic resources used for building local reference databases.
18. **GPU:** The graphics processor targeted for acceleration.
19. **CPU:** The host processor used for analysis when GPU acceleration is unavailable or disabled.
20. **Sequencing Depth:** The total number of reads representing a sample, influencing detection sensitivity.
21. **False Positive:** An incorrect classification where a read is assigned to a taxon not actually present in the sample.
22. **Confidence Score:** A numerical measure indicating the reliability of a taxonomic classification result.
23. **Preprocessing:** Initial analysis steps applied to raw FASTQ data, such as quality filtering and cleanup, before classification.
24. **Deterministic Execution:** Pipeline behavior that produces identical outputs when given the same inputs and reference data.

10 References

[1] IEEE Computer Society, “IEEE Recommended Practice for Software Requirements Specifications,” IEEE Std 830-1998, Oct. 1998.

[2] International Organization for Standardization, *Systems and software engineering—Systems and software Quality Requirements and Evaluation (SQ*International Organization for Standardization, *Systems and software engineering—Systems and software Quality Requirements and Evaluation (SQuaRE)—System and software quality models*, ISO/IEC 25010:2011, 2011*uaRE)—System and software quality models*, ISO/IEC 25010:2011, 2011.

[3] Object Management Group, *Unified Modeling Language (UML) Specification*, Version 2.5.1, Dec. 2017. [Online]. Available: https://www.omg.org/spec/UML/2.5.1

[4] International Organization for Standardization, *Systems and software engineering—Life cycle processes—Requirements engineering*, ISO/IEC/IEEE 29148:2018, 2018.