**High level Architecture**

**P08:AgriQual**

|  |  |
| --- | --- |
| **Student ID** | **Name** |
| **26100370** | **Zarak qadir khan** |
| **26100219** | **syeda umaima hasan** |
| **26100247** | **mishaal usman** |
| **26100259** | **muhammad walid khalid** |

|  |  |  |
| --- | --- | --- |
| **Content** | **Total** | **Obtained** |
| Introduction | 5 | 5 |
| Non-functional requirements | 5 | 5 |
| Security requirements | 5 | 5 |
| Project risk management | 25 | 25 |
| Architecture diagram | 50 | 35 |
| Architecture description | 20 | 20 |
| Architecture justification | 40 | 40 |
| Tools & Technologies | 10 | 10 |
| Hardware Requirements | 5 | 5 |
| Development env. preparation | 10 | 10 |
| Deployment platform | 5 | 5 |
| Use of generative AI | 5 | 5 |
| Who did what | 5 | 5 |
| Review checklist | 5 | 5 |
| Overall formatting | 5 | 2 |
| Late submission penalty | -20 |  |
| **Grand Total** | **200** | **182** |
| **General Comments/Individual Grading:** | | |

**Table of Contents**

[1. Introduction 3](#_gn5x5ivz6dw)

[2. Non-functional requirements/Quality attributes of the system 4](#_2dxby38qjn0)

[3. Security Requirements 5](#_ms7k0pp4r8eh)

[4. Project Risk Management 6](#_naa13fu866o9)

[4.1 Potential Project Risks and Mitigation Strategies 6](#_qts37bjoq0gd)

[5. System Architecture 7](#_xf1c54yxki9p)

[5.1 Architecture Diagram 7](#_up5yso4rlk2a)

[5.2 Architecture Description 8](#_aq3vxmyduhcv)

[5.3 Justification of the Architecture 9](#_561d2ycu48pn)

[6. Tools and Technologies 13](#_2ei7zxasgwwk)

[7. Hardware Requirements 14](#_v71pnu3c182n)

[8. Development Environment Preparation 15](#_g2load1h1htl)

[9. Deployment Platform 15](#_t1f9aseyufhf)

[10. Use of Generative AI 16](#_krlhac5mm4bj)

[11. Who Did What? 17](#_vge48t2slpjv)

[12. Review checklist 17](#_jcyzisb89hvo)

# Introduction

AgriQual is an AI-driven system designed to support farmers by providing personalized, real-time agricultural advice. It uses a multi-agent system, combining Vision, Climate, and Advisory agents to deliver comprehensive solutions aimed at optimizing crop yields, reducing losses, and improving farming efficiency. By leveraging advanced machine learning, farmers gain access to expert-level insights that enhance productivity, sustainability, and profitability.

The Vision Agent analyzes crop images using computer vision techniques to detect diseases, pests, and other indicators of crop health and quality. This allows farmers to monitor crops at different growth stages, enabling early interventions and precise management decisions.

The Climate Agent processes meteorological data from sources to provide accurate weather forecasts and climate risk assessments. This information helps farmers plan irrigation, fertilization, and other activities, while also preparing for long-term environmental risks such as droughts, floods, or frost.

The Advisory Agent integrates insights from the Vision and Climate Agents to generate actionable, personalized recommendations. These recommendations guide farmers on optimal irrigation schedules, pest and nutrient management, and harvest timing, ensuring holistic support for effective decision-making.

AgriQual also benefits quality inspectors by validating crop assessments to meet agricultural standards and supports system administrators in managing infrastructure, monitoring performance, and maintaining data integrity. By making advanced agricultural knowledge accessible, the system empowers farmers to improve efficiency and adopt sustainable practices. Ultimately, AgriQual aims to transform farming into a more data-driven, productive, and profitable activity.

# Non-functional requirements/Quality attributes of the system

|  |  |
| --- | --- |
| 1 | The system should process and classify an uploaded image in less than 15sec |
| 2 | The system should support at least 100 concurrent users |
| 3 | The system should provide a confidence score for each prediction |
| 4 | The system should have a simple and intuitive interface where most user tasks are completed in 2-3 clicks. |
| 5 | The system should handle an increase in dataset size(adding new crops) |
| 6 | The system should provide clear error logs for troubleshooting |
| 7 | The system should retrieve weather data within 30 seconds of a request from the advisory agent to ensure timely recommendations for farming activities |
| 8 | Certificate generation should complete within 5 seconds for single quality assessments |
| 9 | Statistical reports should process 1000 assessment records within 10 seconds. |
| 10 | The system should support crop image uploads up to 10MB. |
| 11 | Notifications should be delivered within 10-20 seconds of system events. |
| 12 | Weather alerts must reach users within 5-10 minutes for severe weather warnings. |

# Security Requirements

|  |  |  |  |
| --- | --- | --- | --- |
| **Sr#** | **Security Risks** | **Potential Losses** | **Controls** |
| 1 | Broken Access Control (e.g., unauthorized user accessing another farmer’s or inspector’s data) | Unauthorized data exposure, privacy violations, reputational damage, possible legal or regulatory penalties. | Apply strict role-based access control (RBAC), verify data ownership before showing or modifying records, and maintain audit logs of access decisions. |
| 2 | Weak or Broken Authentication | Account takeover, fraudulent use of the system, loss of trust if accounts are misused. | Enforce strong password rules, store passwords using hashing (e.g., bcrypt), enable multi-factor authentication (MFA), use session timeouts, and apply login rate-limiting. |
| 3 | Injection Attacks (SQL injection, malicious file input, command injection) | Database compromise, loss of sensitive data, system downtime. | Use parameterized queries, validate all user inputs (IDs, file names, etc.), limit file formats and sizes, and sanitize data before processing. |
| 4 | Data Poisoning (malicious data fed into the diagnosis or training pipeline) | Corrupted crop diagnosis results, wrong recommendations to farmers, potential crop damage, business credibility loss. | Use strict validation of uploaded images, keep a clean and verified training dataset, apply anomaly detection, and log unusual data submissions for review. |
| 5 | Abuse of External Requests (e.g., Weather API misuse, Server Side Request Forgery) | Unauthorized access to internal systems, service disruption, leakage of sensitive information. | Allow only pre-approved external domains, block internal IP ranges, enforce network isolation for API calls, and log all outbound requests. |

# Project Risk Management

## Potential Project Risks and Mitigation Strategies

|  |  |  |
| --- | --- | --- |
| **Sr.** | **Risk Description** | **Mitigation Strategy** |
|  | **Staff turnover**  Experienced staff leaves the project before it is finished. | Maintain detailed project documentation and version control (GitHub) for all modules. Use clear coding standards and frequent knowledge-sharing sessions so that onboarding new members is smooth. Assign backup owners for each module (AI, backend, advisory). |
|  | **Requirements change**  Changes in requirements that require major design rework are proposed. | Use agile development with iterative sprints. Maintain a clear change management process with stakeholder approval. Keep architecture modular so components (e.g., VisionAgent, AdvisoryAgent) can be updated independently. |
|  | **Underestimation**  The size of the system is underestimated. | Include buffer time in each sprint and review project milestones regularly. Use early prototyping to validate complexity. Conduct weekly reviews with advisors and stakeholders. |
|  | **Technology change**  The underlying technology on which the system is built is superseded by a new technology. | Use widely supported frameworks and APIs. Abstract AI agents (VisionAgent, ClimateAgent) behind interfaces to allow model updates. Monitor ML and API versions for compatibility. |
|  | **Code generation**  The code generated by generative AI is inefficient. | Review AI-generated code manually before integration. Use code linting, testing pipelines, and peer review. Train AI tools with project context for higher-quality outputs. |
|  | **Data**  Required data is not available. The required data may be for training of ML Model or for some other purpose. | Build partnerships with agricultural research institutions for dataset access. Use synthetic data generation for missing samples. Maintain data validation scripts and fallback datasets. |
|  | **Stakeholder management**  Customers fail to understand the impact of requirements change. | Conduct regular stakeholder demos and collect feedback. Provide UI mockups and early prototypes. Keep a communication log and maintain simplified documentation for non-technical users. |
|  | **Off the shelf components and libraries**  Software components/libraries that were planned to be used do not contain desired features or contain defects, i.e., they cannot be used as planned. | Evaluate and test all dependencies early. Keep alternative libraries and versions documented. Use Docker or virtual environments to isolate dependencies and avoid conflicts. |

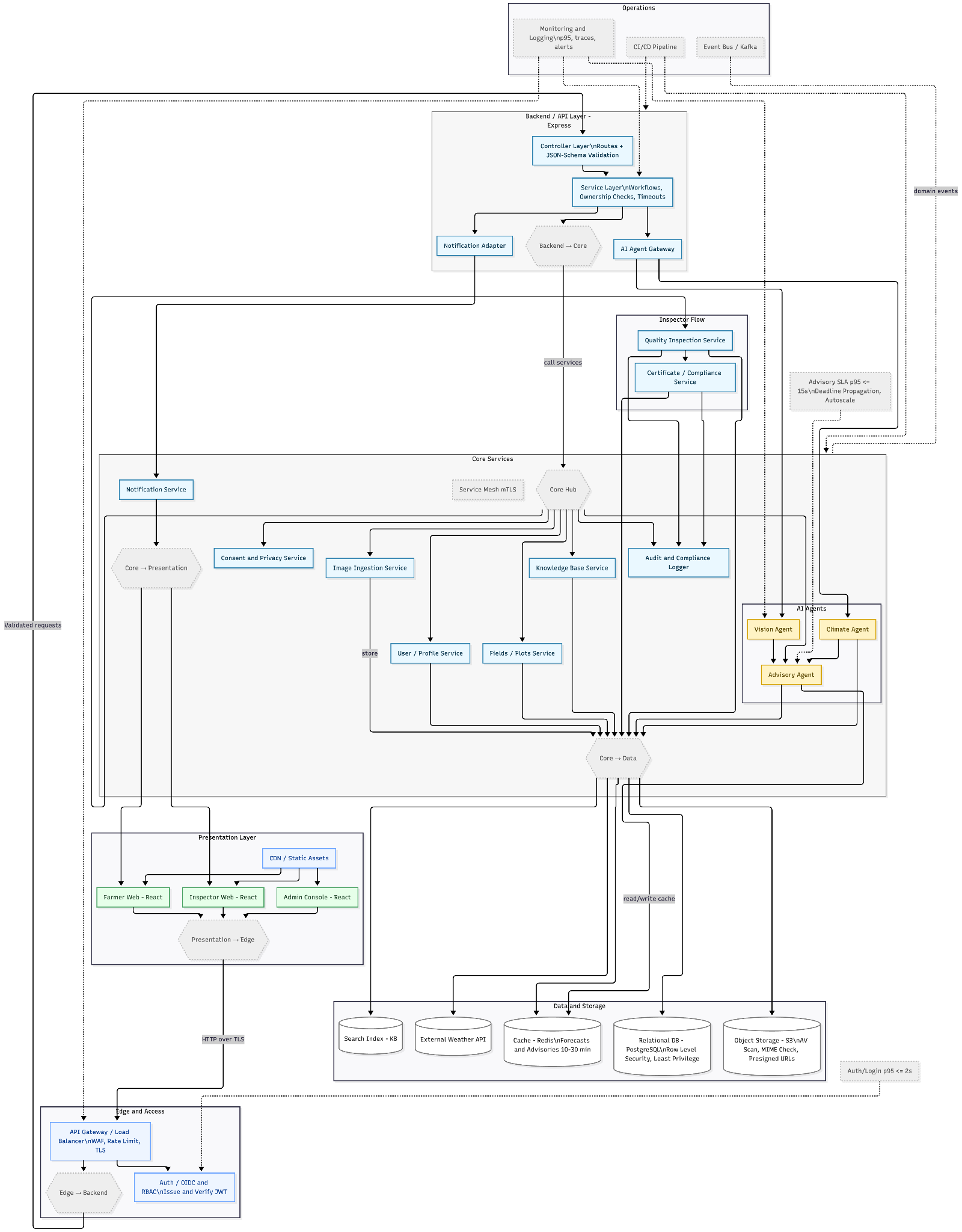
# System Architecture

## Architecture Diagram

Review the layered architecture in the lecture slides, add high-level components in each layer. Follow the pattern of layered architecture like start with the presentation layer. Try to make it more abstract.  
Add some components for performance.

**Layered Architecture:**

(next page)



## Architecture Description

The AgriQual system follows a modular, service-oriented, cloud-native architecture that combines the MERN stack (MongoDB, Express.js, React.js, Node.js) with AI-driven microservices and event-based orchestration. The design supports scalability, high availability, and maintainability while enforcing strict data governance and SLA monitoring.

### **1. Client Layer**

The presentation tier consists of responsive web and mobile interfaces for distinct user roles.

* Farmer App (Mobile/Web):  
   Captures crop photos, GPS location, and crop metadata; displays diagnoses, advisories, and localized weather updates.  
   Utilizes pre-signed URLs for efficient image uploads and receives push/SMS notifications for timely interventions.
* Inspector App (Tablet/Web):  
   Allows inspectors to upload batch images, record grading data, and generate compliance certificates.  
   Supports role-based approvals, overrides, and report exports.
* Admin / Analyst Console:  
   Provides dashboards to manage users, monitor SLAs, update AI models, and review system metrics.  
   Enables audit review, consent management, and model lifecycle control.

### **2.** **Edge and Access Layer**

This layer secures and optimizes access to backend services.

* CDN / Static Assets: Caches static content (JS, CSS, media) to reduce latency.
* API Gateway & WAF: Acts as a unified ingress for client requests, handling routing, rate limiting, and request validation.
* Authentication & RBAC: Implements OAuth2/OIDC standards for token issuance and enforces role-based permissions (Farmer, Inspector, Admin).

Set the formatting of the following sections.

### **3. Core Services & AI Agents**

The core layer encapsulates the main business logic and intelligence services.

* Image Ingestion Service: Handles uploads, performs virus scans and normalization, and stores media in Object Storage (S3).  
   Emits domain events (image\_uploaded) for downstream processing.
* Vision Agent (Inference Service): Uses GPU-optimized models to detect diseases, pests, and severity from uploaded crop images.  
   Ensures SLA compliance (≤15 s per image) using autoscaling and caching.
* Climate Agent / Weather Service: Retrieves real-time weather and satellite data, calculates localized risk indices, and stores time-series data for analytics.
* Advisory / Rules Engine: Combines outputs from Vision and Climate Agents with agronomic knowledge to generate personalized, actionable recommendations.  
   Incorporates PHI/WHI safety, crop calendars, and region-specific rules.
* Quality Inspection Service: Supports inspectors in grading produce, generating remediation advice, and issuing compliance certificates.
* Knowledge Base Service: Hosts FAQs, guides, and contextual help. Integrates with the advisory module for why this advice explanations.
* User & Profile Service: Manages farms, plots, preferences, and notifications.
* Privacy & Consent Service: Logs consent and governs data use for AI training or sharing.
* Notification Service: Sends push, SMS, WhatsApp messages and schedules reminders.
* Audit & Compliance Logger: Maintains immutable trails of system activity.
* SLA Guardrail: Monitors p95 latency and automatically triggers autoscaling or alerts on threshold breaches.

### **4. Data and Storage Layer**

A combination of polyglot storage solutions ensures optimized data management.

* Object Storage: Stores raw images, derived masks, and model artifacts.
* Relational Database (PostgreSQL): Maintains user, advisory, and inspection data.
* Time-Series Database: Logs weather metrics and service performance.
* Cache (Redis): Caches advisories, forecasts, and authentication sessions.
* Search Index: Enables full-text search in the Knowledge Base.
* Feature Store: Provides versioned features for online and batch ML pipelines.
* Data Lake & Warehouse: Stores long-term datasets for analytics and BI.
* Model Registry: Tracks model versions, metadata, and deployment lineage.

### **5. Messaging and Orchestration**

The system relies on asynchronous, event-driven communication for scalability.

* Event Bus (Kafka): Publishes and subscribes to domain events such as image\_uploaded, advisory\_issued, inspection\_submitted.
* Work Queues / DLQ: Handles background processing and retries.
* Workflow Orchestrator: Coordinates multi-step pipelines (e.g., Ingest, Vision, Fuse, Notify).

### **6. ML Ops and Analytics**

Supports the full lifecycle of AI model training and deployment.

* ETL and Training Pipelines: Periodically train models from curated data.
* Experiment Tracking: Logs hyperparameters, datasets, and evaluation metrics.
* Batch Scoring: Runs nightly jobs for regional risk mapping and campaign targeting.
* Canary Rollouts: Gradual deployment of new models with automated rollback on SLA or quality regressions.

### **7. External Integrations**

* Weather APIs: Pull official meteorological data with rate limits and fallback providers.
* Satellite Imagery: Supplies vegetation and cloud data for risk analysis.
* Messaging Gateways: Integrate with email providers

### **8. Runtime Interactions**

Key runtime workflows demonstrate inter-service collaboration:

* **Farmer Advisory Flow:**
  1. Farmer uploads an image → Ingestion Service stores → emits event.
  2. Vision Agent detects disease → Climate Agent enriches with weather → Advisory Engine fuses results.
  3. Advisory stored in DB → cached in Redis → notifications sent → audit logged.
* **Inspector Grading Flow:** Inspector uploads inspection batch → Quality Inspection evaluates → stores in DB and Object Storage → issues certificates.
* **Consent-Aware Processing:** Consent Service validates before any AI training or data sharing.  
   All downstream pipelines respect consent flags.
* **Batch Analytics:** Data Lake processes aggregated insights → triggers retraining when drift is detected → updated models deployed via controlled rollout.

### **9. Security and Operations**

### Security and observability are integral to the architecture. All API calls pass through Auth/RBAC layers; sensitive data is tokenized or minimized. Monitoring dashboards (Grafana/Prometheus) visualize system metrics, while CI/CD pipelines automate testing, deployment, and rollback.

## Justification of the Architecture

* **Pros and cons of the architecture**

|  |  |
| --- | --- |
| **Pros** | **Cons** |
| **Independent Scalability**: Each microservice (Vision Agent, Climate Agent, Advisory Engine) can scale independently based on load, allowing efficient resource allocation when image processing demand spikes while weather data requests remain steady. | **Increased Complexity**: Managing multiple microservices, event buses, and data stores can be complex and difficult to coordinate effectively |
| **Technology Flexibility**: Different services can use optimal technologies—PyTorch for Vision Agent, Node.js for API services, and specialized time-series databases for weather data—without affecting other components. | **Network Latency**: Inter-service communication through API Gateway and event bus introduces network overhead, that we must carefully optimize to meet the 15-second image processing requirement |
| **Fault Isolation**: Failure in one agent (e.g., Weather API timeout in Climate Agent) doesn't crash the entire system; farmers can still receive crop disease diagnoses while weather advisories are temporarily unavailable. | **Data Consistency Challenges**: Maintaining consistency across multiple databases (MongoDB for images, PostgreSQL for users, Redis cache) requires implementing complex synchronization logic that may be difficult to debug. |
| **Role-Based Separation**: Clear service boundaries (Farmer App, Inspector App, Admin Console) with RBAC at the API Gateway level naturally enforce access control and support regulatory compliance for agricultural data | **Testing Overhead**: Comprehensive testing requires unit tests for each service, integration tests across service boundaries, end-to-end tests, significantly increasing QA effort. |
| **Polyglot Persistence**: Using specialized databases (PostgreSQL for relational data, MongoDB for flexible inspection records, Time-series DB for weather, Redis for caching) optimizes performance and storage efficiency for each data type | **Debugging Difficulty**: Tracing requests across multiple services, event streams, and asynchronous workflows makes troubleshooting production issues more complex |
| **MLOps Integration**: Dedicated ML pipeline with model registry, feature store, and canary deployments enables continuous model improvement and safe rollout of updated disease detection algorithms without system downtime. |  |

* **Implementation of non-functional requirements in system architecture**

|  |  |
| --- | --- |
| **1. Requirement** | The system should process and classify an uploaded image in less than 15 seconds. |
| **Implementation in the architecture** | Vision Agent runs on GPU optimized instances with pre-loaded PyTorch models in memory to eliminate initialization delays. Images upload directly to Object Storage via pre-signed URLs, bypassing backend API bottlenecks. Redis caches recent predictions for common crops and diseases, enabling instant responses. SLA Guardrail continuously monitors p95 inference latency and triggers Kubernetes Horizontal Pod Autoscaler when processing time approaches threshold. Multiple Vision Agent containers process images in parallel during peak loads.  **Architectural components:** Vision Agent (GPU instances), Image Ingestion Service, Object Storage, Redis Cache, SLA Guardrail, Kubernetes Autoscaler, Event Bus (Kafka), API Gateway. |

|  |  |
| --- | --- |
| **2. Requirement** | The system should support at least 100 concurrent users. |
| **Implementation in the architecture** | API Gateway with Load Balancer distributes requests across multiple stateless microservice instances in Kubernetes cluster. Session data stored externally in Redis enables any service instance to handle any request. Horizontal Pod Autoscaler monitors CPU/memory utilization and provisions new pods when load exceeds 70% capacity. CDN caches static frontend assets at edge locations, offloading backend traffic. MongoDB and PostgreSQL use connection pooling to handle concurrent queries efficiently. Event Bus decouples synchronous requests from time-intensive operations.  **Architectural components:** API Gateway, Load Balancer, Kubernetes Cluster, Redis (sessions and cache), CDN, MongoDB, PostgreSQL (connection pools), Event Bus (Kafka), Horizontal Pod Autoscaler. |

|  |  |
| --- | --- |
| **3. Requirement** | The system should provide a confidence score for each prediction. |
| **Implementation in the architecture** | Vision Agent's PyTorch model includes softmax output layer generating probability distributions across disease classes. Maximum probability becomes confidence score (0-100%) stored in MongoDB diagnosis collection. Advisory Agent uses confidence scores to determine recommendation reliability—predictions below 70% trigger verification steps or inspector review. Farmer App displays confidence using color-coded visual indicators with explanatory text. Quality Inspection Service flags low-confidence predictions for human validation. MLOps pipeline monitors confidence distributions for drift detection.  **Architectural components:** Vision Agent (PyTorch model), MongoDB (diagnosis collection), Advisory Agent, Quality Inspection Service, Farmer App (React), MLOps Pipeline, Drift Detection, Audit Logger. |

|  |  |
| --- | --- |
| **4. Requirement** | The system should have a simple and intuitive interface where most user tasks are completed in 2-3 clicks. |
| **Implementation in the architecture** | React-based Farmer App implements workflow-optimized design with home screen presenting primary actions as large icon-based buttons. Crop scanning workflow streamlined to 2 clicks: tap button to launch camera, capture photo which auto-uploads and navigates to results. Image Ingestion Service automatically extracts metadata from uploaded images, eliminating manual form fields. User Service maintains farmer preferences in database, pre-populating forms and filtering advisories. React Router implements shallow navigation hierarchy with persistent bottom navigation. CDN ensures fast frontend loading on mobile networks.  **Architectural components:** Farmer App (React), Image Ingestion Service, User & Profile Service, Advisory Agent, React Router, CDN, PostgreSQL (user preferences), API Gateway. |

|  |  |
| --- | --- |
| **5. Requirement** | The system should handle an increase in dataset size (adding new crops). |
| **Implementation in the architecture** | MLOps pipeline separates training infrastructure from production inference services. Expanded training data stored in versioned Object Storage buckets with metadata tracking crop classes. Training Pipeline orchestrated by Airflow runs offline on separate compute instances without affecting production. Feature Store automatically generates and versions features for new crops. Trained models registered in Model Registry and deployed via canary strategy. Vision Agent implements dynamic class loading, reading output classes from configuration files. MongoDB flexible schema accommodates new crop metadata without migrations. Knowledge Base Service stores agronomic rules as JSON files.  **Architectural components:** MLOps Pipeline, Training Pipeline (Airflow), Object Storage (versioned buckets), Feature Store, Model Registry, Vision Agent, MongoDB, Knowledge Base Service, Canary Deployment. |

|  |  |
| --- | --- |
| **6. Requirement** | The system should provide clear error logs for troubleshooting. |
| **Implementation in the architecture** | All microservices emit structured JSON logs with standardized fields (timestamp, service\_name, log\_level, trace\_id, user\_id, error\_code, error\_message). Centralized logging service aggregates logs into searchable system accessible via Grafana dashboards. API Gateway assigns unique trace\_id to each request and propagates through all downstream services, enabling complete distributed request tracing. Each microservice implements detailed error handling with specific error codes. Audit & Compliance Logger maintains immutable event trail. Event Bus includes Dead Letter Queue capturing failed events with full context.  **Architectural components:** Monitoring & Logging Infrastructure (Grafana, Prometheus), API Gateway (trace\_id injection), All Microservices (structured logging), Audit & Compliance Logger, Event Bus (Kafka with DLQ), Centralized Log Aggregation. |

|  |  |
| --- | --- |
| **7. Requirement** | The system should retrieve weather data within 30 seconds of a request from the advisory agent to ensure timely recommendations for farming activities. |
| **Implementation in the architecture** | Climate Agent first checks Redis cache for recent weather data, returning cached results instantly when available. For cache misses, Climate Agent makes parallel asynchronous HTTP requests to multiple Weather API providers using asyncio library, accepting first successful response. Connection pooling to weather endpoints reduces network overhead. Rate-limiting middleware prevents quota exhaustion at weather providers. Aggressive timeout configuration (10 seconds max per API request) ensures worst-case response within SLA. Event Bus enables asynchronous processing where Advisory Agent continues other work while waiting for weather response..  **Architectural components:** Climate Agent, Redis Cache, Weather API Integration, Event Bus (Kafka), SLA Guardrail, Advisory Agent, Secrets Manager (API keys). |

|  |  |
| --- | --- |
| **8. Requirement** | Certificate generation should complete within 5 seconds for single quality assessments. |
| **Implementation in the architecture** | Quality Inspection Service uses pre-loaded PDF templates stored in memory at service startup rather than fetching per request. Inspection data stored in PostgreSQL with indexed fields for rapid retrieval. Lightweight templating engine populates certificate fields efficiently. Generated certificates immediately stored in Object Storage with metadata saved to the database. Service runs multiple instances behind Load Balancer to handle concurrent certificate requests. Pre-signed URLs provide instant download access without additional processing.  **Architectural components:** Quality Inspection Service, PostgreSQL (indexed tables), Object Storage (certificates), Load Balancer, Redis (template cache), API Gateway, Pre-signed URL Generator. |

|  |  |
| --- | --- |
| **9. Requirement** | Statistical reports should process 1000 assessment records within 30 seconds. |
| **Implementation in the architecture** | Quality Inspection Service retrieves assessment records from PostgreSQL using optimized queries with composite indexes on commonly filtered fields. Report generation runs on dedicated service instances with sufficient memory allocation for large dataset processing. Batch processing techniques aggregate 1000 records without loading the entire dataset into memory simultaneously. Redis caches frequently requested report data providing instant results for common queries. Report Generation module uses efficient data processing libraries (Pandas, NumPy) for rapid calculations. Database query planner optimizes execution with proper indexing.  **Architectural components:** Quality Inspection Service (dedicated instances), PostgreSQL (indexed, optimized queries), Redis (report cache), Report Generation Module (Pandas/NumPy), Load Balancer, Object Storage (report archives), Time-series Database. |

|  |  |
| --- | --- |
| **10. Requirement** | The system should support crop image uploads up to 2MB. |
| **Implementation in the architecture** | Image Ingestion Service validates uploaded file sizes before processing, rejecting files exceeding 10MB with clear error messages. Pre-signed URLs from Object Storage allow farmers to upload images directly, bypassing the backend API. API Gateway configured with appropriate request size limits to handle 10MB payloads safely. Object Storage configured for multi-part uploads, enabling large file uploads reliably on slower mobile networks. Frontend performs client-side file size validation before upload. Image Ingestion Service processes uploaded files with virus scanning asynchronously without blocking user workflow.  **Architectural components:** Image Ingestion Service (with virus scanning), Object Storage (pre-signed URLs, multi-part upload), API Gateway (size limits), Farmer App (client validation), CDN, Event Bus (async processing). |

|  |  |
| --- | --- |
| **11. Requirement** | Notifications should be delivered within 10-20 seconds of system events. |
| **Implementation in the architecture** | Notification Service subscribes to relevant events from Kafka Event Bus, receiving real-time notifications when advisories generated or diagnoses completed. Service maintains persistent connections to notification gateways (SMS, FCM, APNs) eliminating connection setup time. Notifications queued in Redis for reliable delivery with exponential backoff retry mechanisms for failures. Service processes notifications asynchronously, preventing delays in primary application workflows. Multiple Notification Service instances run in parallel to handle high volumes of simultaneous notifications. Notification delivery tracked in Audit Logger with timestamps for SLA monitoring.  **Architectural components:** Notification Service (multiple instances), Event Bus (Kafka), Redis (notification queue), SMS/Push Gateways (FCM, APNs), Load Balancer, Audit Logger, Dead Letter Queue. |

|  |  |
| --- | --- |
| **12. Requirement** | Weather alerts must reach users within 5-10 minutes for severe weather warnings. |
| **Implementation in the architecture** | Climate Agent continuously polls Weather APIs for severe weather updates using scheduled jobs. Urgent alerts published to Event Bus with high priority flags for immediate processing. Notification Service subscribes to weather alert events with dedicated processing threads for urgent messages. Severe weather notifications bypass normal queuing and sent immediately through multiple channels (SMS, push notifications) simultaneously. Service maintains location-based user registry to target affected farmers specifically. Alert delivery tracked in Audit Logger with timestamps to monitor SLA compliance.  **Architectural components:** Climate Agent (polling service), Weather API Integration, Event Bus (Kafka), Notification Service (dedicated threads), SMS Gateway, Push Notification Service (FCM/APNs), User & Profile Service (location registry), Audit Logger, Secrets Manager. |

* **Implementation of security requirements in system architecture**

|  |  |
| --- | --- |
| Security Risk | **Broken Access Control**  Broken access control typically leads to unauthorized information disclosure, modification, or destruction of all data or performing a business function outside the user's limits. |
| Implementation in the architecture | A user logs in via the Web Frontend. Credentials are validated by the Auth/User-Management Service. On success, the service issues short-lived access tokens (JWT). The Web Frontend shows only features allowed by the user’s role. All API calls go through the API Gateway, which validates tokens and forwards requests only to authorized microservices in a private network. Each microservice re-checks ownership/tenant before data read/write. All API traffic is encrypted in transit.  Architectural components: Web Frontend, Auth/User-Management Service, API Gateway (with WAF/rate limits), Business Microservices (Farmer/Inspector/Diagnosis/etc.) in Private VPC/Subnets, Service Mesh (mTLS), Relational DB with Row-Level Security, Audit/Logging Service. |

|  |  |
| --- | --- |
| Security Risk | **Injection Attacks (SQL injection, file/command injection)**  Untrusted input can alter queries or invoke unintended commands, leading to data compromise or remote execution. |
| Implementation in the Architecture | The API Gateway enforces OpenAPI/JSON-Schema validation before requests reach services. Microservices use parameterized queries/ORM only and least-privilege DB accounts. File uploads go through an Ingestion Service using pre-signed URLs; files are AV-scanned, MIME/magic-byte checked, size/extension-limited, and stored with randomized keys. Services avoid shelling out; if needed, tasks run in sandboxed containers. Errors are normalized (no stack traces). All traffic is encrypted.  Architectural components: API Gateway (WAF/request validation), Ingestion Service, Object Storage, Business Microservices, Container Sandbox/Worker Pool, Relational DB, Service Mesh (mTLS). |
| Security Risk | **Data Poisoning (tainted inputs into diagnosis/training)**  Malicious or low-quality data can corrupt model training or skew diagnosis, harming recommendations and trust. |
| Implementation in the Architecture | Uploads are admitted only via the Ingestion Service with quality, format, and anomaly checks; suspicious samples are quarantined. Production inference uses models from a signed, versioned Model Registry. Training reads only approved datasets from a Dataset Registry with provenance and checksums. New models are canary-deployed and monitored for performance/drift; automatic rollback on KPI degradation. All steps are auditable. Architectural components: Ingestion Service, Object Storage, Dataset Registry, Model Registry, ML/Inference Service, CI/CD (Canary), Monitoring/Drift Detection, Audit/Logging Service. |
| Security Risk | **Abuse of External Requests / SSRF (e.g., Weather API misuse)**  Abused outbound calls can reach internal resources, exfiltrate data, or incur costs. |
| Implementation in the Architecture | Services never call user-provided URLs. All egress goes through an Egress Proxy with strict domain/IP allow-lists; private/address-metadata ranges are blocked. A dedicated Weather/External-Fetch Service runs in an isolated subnet with no lateral access. Outbound requests are rate-limited, time-bounded, size-capped, and fully logged. API keys are scoped and stored in the Secrets Manager.  Architectural components: Egress Proxy (allow-list, RFC1918 deny), Weather/External-Fetch Service (isolated subnet), API Gateway, Service Mesh (mTLS), Secrets Manager (scoped API keys), Monitoring/SIEM, Private VPC/Subnets. |
| Security Risk | **Sensitive Data Exposure**  PII, field locations, images, or inspection results could leak in transit, at rest, via logs, or misconfigured sharing. |
| Implementation in the Architecture | TLS 1.2+ for client↔Gateway and mTLS for service↔service. At-rest encryption for DB, object storage, backups, and model/artifact stores with KMS-managed keys. Data minimization between services; logs redact tokens/PII. Highly sensitive fields use application-level encryption; access bound to roles and purpose. Exports use time-bound pre-signed URLs; CORS is restricted. Access to data is monitored and audited with regular reports.  Architectural components: API Gateway, Service Mesh (mTLS), Relational DB (encrypted/RLS), Object Storage (encrypted), KMS/Key Management, Auth/User-Management Service, Audit/Logging Service, Monitoring/SIEM, Private VPC/Subnets. |

# Tools and Technologies

### **6.1 Development Tools**

* Visual Studio Code (1.92): Primary IDE for frontend, backend, and AI modules.
* JavaScript (ES2023 / ECMAScript 14): Core language for both React and Node.js logic.
* Postman (11.0): API design, testing, and documentation.
* Git (2.45) + GitHub (Enterprise Cloud): Version control, collaboration, and CI/CD integration.
* Node.js (20.x) + npm (10.x): Backend runtime and dependency management.
* Python (3.11): AI/ML model development, data processing, and automation scripts.
* Jupyter Notebook (7.x): Interactive environment for experimentation and analysis.

### 

### **6.2 Frontend**

* React.js (18.x): Dynamic web UI for farmers, inspectors, and admins.
* Tailwind CSS (3.4): Utility-first styling for responsive, clean UIs.
* Axios (1.6): Handles secure API calls with token-based authentication.
* React Router (6.23): Client-side routing and navigation control.
* Firebase Cloud Messaging (12.0): Push notifications and alerts delivery.

### **6.3 Backend / APIs**

* Express.js (4.19): RESTful API framework for Node.js services.
* FastAPI (0.110): Python-based framework for high-performance AI service endpoints.
* MongoDB (7.0) : NoSQL database for advisory logs and image data.
* PostgreSQL (16.3) : Relational database for users, inspections, and consent records.
* Redis (7.2) : In-memory cache for sessions, weather data, and quick responses.

### **6.4 AI / Machine Learning**

* PyTorch (2.3) : Model training and inference for crop disease detection.
* OpenCV (4.9) : Image processing and computer vision utilities.
* scikit-learn (1.5) : Auxiliary ML models and classification support.
* Pandas (2.2) : Data manipulation and analysis.
* NumPy (1.26) : Numerical computations and feature generation.
* TensorBoard (2.17) : Model performance monitoring and visualization.

### **6.5 Security & Compliance**

* JWT (2.9) : Stateless authentication for user sessions.
* AWS KMS (2025-Q1) : Encryption key management.
* OAuth2 / OIDC : Federated identity and role-based access control.

### **6.6 Deployment / Cloud Infrastructure**

* AWS EC2 (t3.medium) : Compute instances for backend microservices.
* AWS S3 (2025-Q1) : Object storage for images, models, and backups.
* Docker (27.x) : Containerization for consistent deployment across environments.
* GitHub Actions : CI/CD pipelines for automated build and deployment.
* Grafana + Prometheus : Monitoring, alerting, and service observability.
* Kafka (3.7) : Event bus for asynchronous inter-service communication.

### **6.7 Testing & Quality Assurance**

* Jest (29.7) : Unit testing for React components.
* PyTest (8.3) : Testing framework for backend and ML services.
* Postman Collections (11.x) : Automated API testing and integration validation.
* ESLint (9.x) + Prettier (3.x) : Code linting and style consistency.

# Hardware Requirements

**A. Development Machines**

Hardware specifications required for developers, data scientists, and testers working on AgriQual.

* Processor: Intel Core i7 (12th Gen or higher) / AMD Ryzen 7 (8 cores, 16 threads, ≥2.8 GHz)
* Memory (RAM): Minimum 16 GB (Recommended: 32 GB for AI model training and Docker multitasking)
* Storage: 512 GB SSD (Recommended: 1 TB NVMe SSD for datasets and containers)
* GPU (for AI module developers)**:** NVIDIA RTX 3060 or higher with 6 GB+ VRAM (for Vision Agent training/inference)

**Operating System:**

* Windows 11 Pro (64-bit) — primary development environment
* Ubuntu 22.04 LTS (via WSL2 or dual boot) — for testing containerized services and ML pipelines

**B. Deployment Servers / Cloud Infrastructure**

Hardware (or equivalent cloud specs) required for hosting the AgriQual platform and running production workloads.

**1. Application & API Layer (Kubernetes Cluster)**

* Compute Nodes:
  + 3 × t3. medium instances (2 vCPUs, 4 GB RAM each) for microservices and gateway

**2. AI Inference / Vision Agent Nodes**

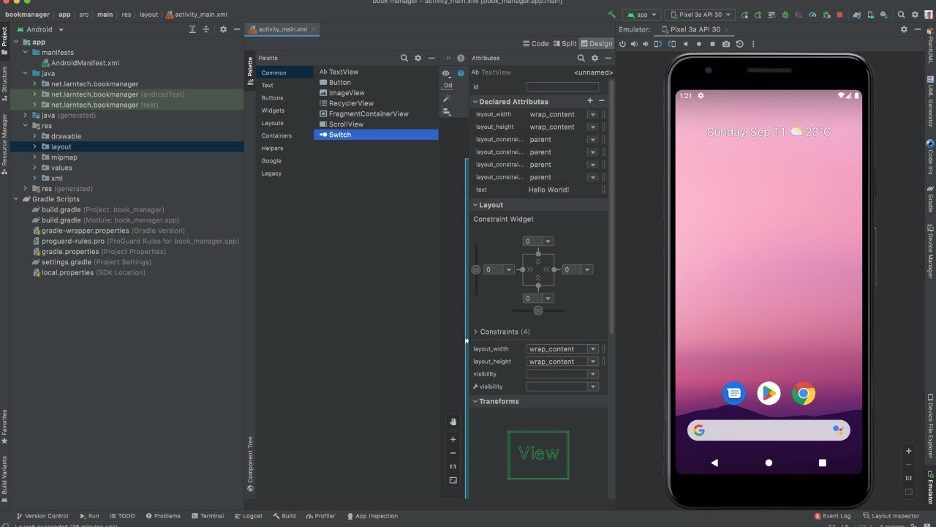
* Compute Nodes:
  + 2 × g4dn.xlarge (4 vCPUs, 16 GB RAM, NVIDIA T4 GPU 16 GB VRAM)
* Purpose: Real-time disease/pest image classification under 15 seconds latency SLA
* Storage: 200 GB EBS SSD for model artifacts and temporary image caching

**3. Database & Storage Layer**

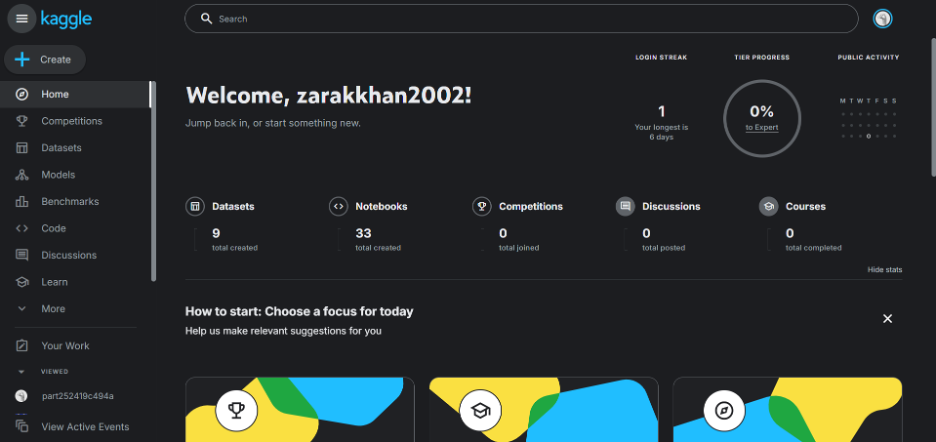
* Relational Database Server (PostgreSQL RDS):
  + 2 vCPUs, 8 GB RAM, 100 GB SSD storage (auto-scaling enabled)
* Object Storage (S3):
  + 1 TB bucket for crop images, inspection photos, and ML artifacts
* Redis Cache Server:
  + 2 vCPUs, 4 GB RAM (Elasticache) for session and advisory caching

# Development Environment Preparation

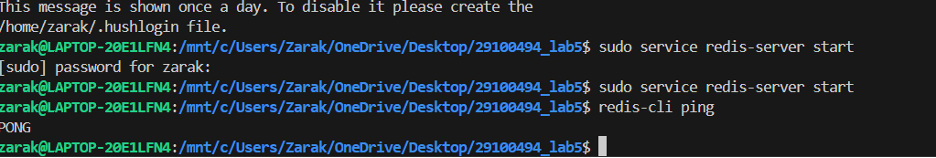
**Android studio:**



**Kaggle:**

****

**Redis server working on the laptop locally:**



# Deployment Platform

AWS Free Tier will be used for deployment of prototype, sprints, and final system. This provides free access to EC2 compute instances for hosting microservices, S3 object storage for images and model artifacts, and managed services for databases. AWS supports our complete architecture including GPU instances for the Vision Agent, managed Kubernetes for container orchestration, and all required infrastructure components specified in the architecture.

# 

# 

# Use of Generative AI

Generative AI was used to rephrase the content provided so that it was in an easy to understand and simplified text. It was also used to cross check definitions of state, class and sequence diagrams to ensure proper and correct diagrams were made and submitted.

# 

# Who Did What?

|  |  |
| --- | --- |
| **Name of the Team Member** | **Tasks done** |
| zarak | downloaded softwares, hardware requirements |
| umaima | tools and technologies, security requirements and implementation |
| mishaal | system architecture, project risk management |
| walid | justification of architecture + the rest of the doc |

# Review checklist

Before submission of this deliverable, the team must perform an internal review. Each team member will review one or more sections of the deliverable.

|  |  |
| --- | --- |
| **Section** **Title** | **Reviewer Name(s)** |
| downloaded softwares, hardware requirements | umaima |
| tools and technologies, security requirements and implementation | mishaal |
| system architecture, project risk management | walid |
| justification of architecture + the rest of the doc | zarak |