



XBCAD7319

POE Document



SEPTEMBER 27, 2024
RIHLAZANA AUCTIONS MOBILE APP

Table of Contents

1. Introduction to the Project	3
1.1 Work Agreement.....	3
1.2 Definition of Ready (DoR)	4
1.3 Definition of Done (DoD)	5
1.4 Roadmap (High-level Plan).....	5
2. Requirements.....	7
2.1 User Roles	7
2.2 User Stories	7
2.3 User Experience Journey Map	11
3. Non-functional Requirements	14
4. Analysis Artifacts	18
4.1 Domain Modelling.....	18
5. Design Artifacts	19
6. Implementation Documentation	20
6.1 Lot Creation Flow	20
6.2 Critical Implementations.....	21
6.2.1 Image Processing Service	21
6.2.2 PDF Catalog Generation Service	21
6.2.3 Mobile App Navigation Implementation	22
6.3 Core Data Models	22
6.3.1 Lot Data Transfer Object (DTO).....	22
6.4 Repository Implementation	23
6.5 Error Handling Implementation	24
6.6 Security Implementation	24
6.7 Testing Implementation.....	24
7. Data Schema Documentation	26
8. Architecture Artifacts.....	30
8.1 Design Patterns	30
8.2 Architecture Patterns.....	30
8.3 Cloud	30
9. Security	35
10. DevOps	38
10.1 GitHub Actions Pipeline	38
11. Running Costs.....	40
○ 11.1 Predicted User Growth	40

- 11.2 Scaling Points for AWS Services40
- 11.3 Predictive Models (Monthly Basis over Two Years).....40
- **Best Case Scenario (High Growth)**40
- **Worst Case Scenario (Low Growth)**40
- **Average Growth Scenario**41
 - 11.4 Technology Adjustments at Scale41
- 12. Change Management.....42
 - Organizational Adoption**42
 - User Adoption**43
 - Adoption and Support Strategy**43
- References44

1. Introduction to the Project

Rihlazana Auctions is an established auction company facing operational challenges in their lot cataloguing process. The current system relies on manual processes and informal communication channels, leading to inefficiencies and potential errors. Our team has been tasked with developing a mobile application to streamline and automate these processes, improving operational efficiency and enhancing the user experience for both the auction company staff and potential bidders.

The primary issues identified are:

1. The operations team needs an efficient way to capture and organize pictures of auction lots (e.g., cars).
2. The current cataloguing process is manual and time-consuming, involving sharing photos via WhatsApp and manually creating PDF catalogues.

Our solution aims to address these issues by developing a mobile application with the following key features:

1. Automated photo upload system with specific formatting requirements.
2. Catalogue builder to organize lots and assign numbers.
3. Automated PDF generation for lot lists.
4. Secure login system for operations team/admin and users.
5. Integration with Rihlazana Auctions' social media platforms via API.

This project presents unique challenges in terms of data management, user experience design, and system integration. We will need to carefully consider security, scalability, and ease of use throughout the development process.

1.1 Work Agreement

As a team, we have agreed to the following principles to guide our collaboration throughout this project:

1. Communication: We will hold meetings at 7:00 PM via Microsoft Teams. All team members are expected to attend and provide updates on their progress, plans and any blockers.
2. Task Management: All team members are responsible for updating their task statuses daily.
3. Code Reviews: All code changes must be reviewed by at least one other team member before merging into the main branch. Code reviews should be completed within 24 hours of submission.

4. Documentation: All major decisions, architecture changes, and important discussions should be documented in our shared Google Drive folder.
5. Deadlines: We commit to meeting all sprint deadlines. If a team member foresees any issues with meeting a deadline, they should communicate this to the team as soon as possible.
6. Respect: We will maintain a respectful and professional environment, valuing diverse opinions and constructive feedback.
7. Continuous Learning: We encourage sharing knowledge and learning from each other. If a team member discovers a useful resource or technique, they should share it with the team.
8. Work-Life Balance: We respect each other's time outside of work hours. Non-urgent communications should be limited to agreed-upon work hours.
9. Conflict Resolution: In case of disagreements, we will first attempt to resolve them among ourselves. If unable to reach a resolution, we will involve our project mentor for mediation.
10. Quality: We commit to writing clean, well-documented code and following best practices in software development.

1.2 Definition of Ready (DoR)

A user story is considered ready for implementation when it meets the following criteria:

1. The story is written in the standard format: "As a [user role], I want [goal/desire] so that [benefit]."
2. The story has been discussed and understood by the entire team.
3. Acceptance criteria are clearly defined and agreed upon.
4. The story has been estimated by the team and fits within a single sprint.
5. Any external dependencies have been identified and resolved.
6. Necessary design assets or mock-ups are available.
7. The story aligns with the project's overall goals and requirements.
8. Performance criteria, if applicable, have been defined.
9. Security implications have been considered and documented.
10. The story has been prioritized by the Product Owner.

1.3 Definition of Done (DoD)

A user story is considered done when it meets the following criteria:

1. Code has been written that fulfils all acceptance criteria.
2. Unit tests have been written and all tests passes.
3. Code has been reviewed and approved by at least one other team member.
4. The feature has been tested on both Android and iOS platforms.
5. Documentation has been updated, including inline code comments and README files.
6. The feature has been demonstrated to and accepted by the Product Owner.
7. The code has been merged into the main branch.
8. The feature passes all automated CI/CD checks.
9. Performance tests have been conducted and meet the defined criteria.
10. A security review has been completed.
11. The feature has been successfully deployed to the staging environment.
12. Any bugs or issues discovered during testing have been resolved.

1.4 Roadmap (High-level Plan)

Our project will be divided into 4 two-week sprints, totalling 8 weeks of development time. Here's our high-level roadmap:

Sprint 1: Project Setup and Requirements Gathering

- Set up development environment and tools
- Conduct detailed requirements gathering sessions with Rihlazana Auctions
- Define user roles and create initial user stories
- Create project architecture diagram

Sprint 2: Core Functionality Development

- Develop user authentication system
- Implement photo upload functionality with specific formatting requirements

- Begin development of catalogue organization feature

Sprint 3: Catalogue and PDF Generation

- Complete catalogue organization feature
- Develop automated PDF generation for lot lists
- Start implementation of admin dashboard

Sprint 4: User Interface and Experience

- Complete admin dashboard
- Develop user interface for browsing lots
- Implement search and filter functionality for lots

Sprint 5: API Integration and Testing

- Integrate with Rihlazana Auctions' social media platforms via API
- Conduct thorough testing of all implemented features
- Begin performance optimization

Sprint 6: Security and Performance Optimization

- Implement additional security measures
- Continue performance optimization
- Conduct user acceptance testing

Sprint 7: Final Testing and Deployment Preparation

- Resolve any remaining bugs or issues
- Prepare deployment documentation
- Conduct final round of testing
- Prepare for app store submission

2. Requirements

2.1 User Roles

1. Operations Team Member: Responsible for capturing and uploading photos of auction lots, creating catalogues, and managing auction information.
2. Administrator: Has overall control of the system, can manage user accounts, view analytics and oversee all operations export the images and PDF catalogues .
3. Potential Bidder: Can view upcoming auctions, browse lot catalogues, and access basic information about lots.
4. Guest User: Can view limited information about upcoming auctions without logging in.

2.2 User Stories

1. As an Operations Team Member, I want to upload multiple photos for a lot in a single action so that I can efficiently catalogue items.
2. As an Operations Team Member, I want the app to automatically create a ZIP folder for each lot so that photos are organized consistently.
3. As an Operations Team Member, I want the app to adjust photo aspect ratios and quality automatically so that all images in the catalogue are uniform.
4. As an Administrator, I want to be able to generate a PDF catalogue of lots with a single click so that I can quickly produce documentation for auctions.
5. As a Potential Bidder, I want to browse through upcoming auction lots on my mobile device so that I can plan my bidding strategy.
6. As a Guest User, I want to view basic information about upcoming auctions without logging in so that I can decide if I want to register.
7. As an Administrator, I want to be able to post auction updates to our social media accounts directly from the app so that I can keep our audience informed efficiently.
8. As an Operations Team Member, I want to be able to edit lot information after upload so that I can correct any mistakes or add additional details.
9. As a Potential Bidder, I want to save lots I'm interested in so that I can easily find them later.
10. As an Administrator, I want to view analytics on user engagement with different lots so that I can better understand bidder interests.

2.1 User Roles

1. Operations Team Member

- **Responsibilities:**
 - Captures and uploads photos of auction lots.
 - Adds detailed information about each lot (e.g., name, quantity, condition).
 - Creates catalogues and submits them for admin approval.
 - Organizes lots by assigning appropriate lot numbers, ensuring each lot's images and details are accurately entered.
- **Permissions:**
 - Upload images and enter data.
 - Access their own uploads and submissions.
 - Modify or delete entries before submission.
 - View upcoming auctions and catalogues they helped create.

2. Administrator

- **Responsibilities:**
 - Has overall control of the system.
 - Manages user accounts (add, modify, delete).
 - Reviews, approves, or rejects catalogue submissions from the operations team.
 - Generates and exports PDFs of catalogues.
 - Manages integrations with social media platforms for sharing auction details.
 - Views system analytics (e.g., how many lots were uploaded, catalogue performance, number of views by potential bidders).
 - Oversees and manages the creation and distribution of ZIP files containing images of lots for marketing purposes.
- **Permissions:**
 - Full access to the system, including adding, editing, or deleting any user's content.
 - Review and approve catalogues before distribution.
 - Access to reporting and analytics data.

- Manage system settings, such as catalogue template designs or compression ratios for images.

3. Potential Bidder

- **Responsibilities:**
 - Can view detailed information about upcoming auctions.
 - Browse full auction lot catalogues.
 - Access detailed descriptions and images of individual lots.
- **Permissions:**
 - View upcoming auction schedules.
 - Access full catalogues, including all lot images and descriptions.
 - Save or download catalogues

2.2 User Stories

User Stories outline how each user type interacts with the system in real-world scenarios. Below are examples of user stories for each role.

1. Operations Team Member

- *As an operations team member, I want to upload photos of auction lots and add detailed descriptions, so that the administrator can create a professional catalogue.*
- *As an operations team member, I want to be able to correct or update lot information before the admin approves it, so that errors can be fixed quickly.*

2. Administrator

- *As an administrator, I want to review and approve submitted catalogues, so that only accurate and complete catalogues are made available to potential bidders.*
- *As an administrator, I want to export PDFs of auction catalogues and share ZIP folders of images, so that marketing efforts are streamlined.*

3. Potential Bidder

- *As a potential bidder, I want to view detailed information and images of lots, so that I can make an informed decision about attending an auction.*
- *As a potential bidder, I want to be able to download catalogues, so that I can review them offline and plan my bids.*

2.3 Functional Requirements

2.3.1 Photo Upload and Catalogue Creation

- Operations team members can upload images of auction lots via the app or web platform.

- The system will accept multiple image uploads for a single lot (e.g., up to 10 images per lot).
- The system will automatically resize and compress images to maintain consistent quality while saving space.
- Lots will be catalogued with relevant data, including:
 - Lot number.
 - Item name.
 - Quantity.
 - Condition.
 - Description (optional).
- Administrators will be able to review, edit, and approve catalogues for publication.

2.3.2 PDF Generation

- The application will generate a PDF catalogue that organizes lots by their assigned lot numbers.
- The PDF will include:
 - Lot details (name, quantity, condition, etc.).
 - Photos of each lot.
 - Auction event information (e.g., date, location, terms).
- Admins can download or distribute PDF catalogues for each auction.

2.3.3 Social Media Integration

- The system will integrate with major social media platforms via API.
- Administrators can post individual lots or entire auction catalogues to social media for marketing.
- Users (operations and admins) can configure and schedule posts directly from the app.

1. User Sign-In/Sign-Up Flow

- **Flow 1: Guest User Access**
 - Users who do not have accounts can access limited functionalities by entering the app directly without signing in.
 - This option leads directly to an onboarding or introduction screen.
- **Flow 2: Sign-In**
 - Registered users (Operations Team Members, Administrators, or Potential Bidders) can log in using their credentials.
 - The sign-in page provides options for social media login (e.g., Google, Facebook) and traditional email/password login.
 - Upon successful login, the user proceeds to the onboarding screen.
- **Flow 3: Sign-Up**
 - New users can register for an account (reserved for Operations Team Members, Administrators, or Potential Bidders).
 - After filling in their details, they will be taken to the onboarding screen, where they are introduced to the app's functionalities.

2. Onboarding Flow

- The onboarding screen provides new users with an overview of the app's primary features, such as:
 - Uploading and managing lots.
 - Creating PDF catalogues.
 - Viewing upcoming auctions and detailed lot information.
 - Connecting with auction platforms and social media.
- After completing the onboarding process, users are taken to the auction home screen.

3. Auction Overview Screen

- The auction overview screen is the main dashboard for users.
- **Administrators and Operations Team Members:**
 - They can manage upcoming auctions, view existing catalogues, and add new auction lots.
- **Potential Bidders and Guest Users:**
 - They can view the auction schedule and basic details of available lots (limited for guest users).

4. Lot Creation Flow

- **Lot Overview Screen:**
 - Operations team members can view a list of auction lots.
 - They can select a lot to view or create a new lot.
- **Camera & Upload Screen:**
 - From the lot screen, the operations team can take photos of individual items.
 - They can upload multiple images for a single lot and add specific details, such as:
 - Lot Name.
 - Quantity.
 - Condition.
- **Editing Screen:**
 - After uploading, the user can review the images, adjust details, and finalize the lot information before saving.
 - The app will automatically organize the images and create a ZIP file based on predefined aspect ratios and quality.

5. Catalogue Generation Flow

- After the lots are created and reviewed, the system generates a PDF catalogue for the auction.
- **PDF of Lots:**
 - The PDF includes images and detailed information (name, condition, quantity) for each lot.
 - The app arranges the images and details according to the auction's needs.

6. Exporting & Sharing

- **Export Screen:**
 - The operations team or administrator can export the catalogue as a PDF or ZIP file containing all images and data.
 - There's an option to save the files for easy sharing or integration with social media platforms.
- **File Viewing Screen:**
 - Users (admins or operations) can view the saved files, review, or download .

7. Viewing Auctions & Catalogue (for Bidders)

- **Auction Viewing Screen:**
 - Bidders can view the upcoming auctions and access the PDF catalogue.

- They can browse the auction lots, view detailed information, and decide whether to participate in the auction.

3. Non-functional Requirements

1. Performance:

- The app should load within 3 seconds on a 4G connection.
- Photo uploads should complete within 5 seconds for a set of 10 photos on a 4G connection.
- PDF generation should complete within 10 seconds for a catalogue of up to 100 lots.

2. Scalability:

- The system should be able to handle up to 1000 concurrent users without performance degradation.
- The database should be able to store up to 100,000 lots with associated photos without significant query time increases.

3. Reliability:

- The app should have an uptime of 99.9%.
- In case of failure, the system should be able to recover within 5 minutes.

4. Maintainability:

- The codebase should follow clean code principles and be well-documented to allow for easy maintenance and updates.
- The system should use modular architecture to allow for easy feature additions or modifications.

5. Security:

- All data transmissions should be encrypted using TLS 1.3.
- User passwords should be hashed and salted before storage.
- The app should implement multi-factor authentication for admin accounts.

6. Usability:

- The user interface should be intuitive and require no more than 3 clicks to reach any feature.
- The app should be accessible, following WCAG 2.1 AA standards.

7. Interoperability:

- The app should be able to integrate with major social media platforms (Facebook, Twitter, Instagram) via their respective APIs.
- The system should be able to export data in common formats (CSV, JSON) for use in other systems.

8. Internationalization/Localization:

- The app should support multiple languages, initially English and Afrikaans.
- The app should handle different date and currency formats based on the user's locale.

Performance

- **Response Time:** The system must upload images, create catalogue entries, and generate PDF files within 2 seconds per lot.
- **Scalability:** The application must handle the concurrent uploading of lots from up to 50 operations team members without degradation in performance.
- **Image Upload Speed:** The system should process a batch of 10 high-resolution images within 5 seconds over a 5Mbps network.
- **Load Handling:** The system should support up to 500 simultaneous users, including guest users browsing auction lots, without experiencing latency issues.

Security

- **User Authentication:** All users must authenticate using OAuth or a secure username/password system. Data for each role (Operations Team, Administrator, Potential Bidders) must be protected.
- **Data Encryption:** All data (images, auction lots, PDFs) must be encrypted during transmission (using SSL/TLS) and at rest in the cloud storage.
- **Access Control:** Role-based access control must be implemented to ensure that only authorized users (Operations Team and Administrators) can modify, upload, or delete auction lots.

- **Data Privacy:** Adhere to data protection regulations like POPIA (Protection of Personal Information Act) for South Africa, ensuring that personal user data (especially for bidders) is protected.

Availability and Reliability

- **System Uptime:** The app must be available 99.9% of the time, allowing users (Operations Team, Administrators, and Bidders) to access it around the clock.
- **Backup and Recovery:** Regular backups of auction data, catalogues, and images must be performed every 24 hours, with a recovery time of less than 30 minutes in case of failure.
- **Fault Tolerance:** The system should automatically recover from common failures (e.g., server downtime) without data loss.

Usability

- **Ease of Use:** The user interface must be simple, with intuitive navigation for both the Operations Team and Potential Bidders. Users should be able to upload images and create auction lots with minimal effort.
- **Responsiveness:** The app should be responsive on multiple platforms, including mobile (iOS and Android), tablets, and desktop browsers. The design must follow mobile-first principles.
- **Onboarding:** First-time users should be able to understand and use the app without requiring extensive documentation or training, thanks to guided onboarding and tooltips.
- **Multilingual Support:** The app must support at least two South African languages, in addition to English, to accommodate a diverse user base.

3.5 Maintainability

- **Modular Architecture:** The system should be built using a modular architecture (e.g., microservices) to allow easy maintenance and feature updates without disrupting the entire system.
- **Code Quality:** Adhere to clean coding practices, with code reviewed regularly for maintainability and clarity. Continuous Integration (CI) and Continuous Delivery (CD) pipelines must be implemented to ensure rapid but stable updates.
- **Testability:** Unit tests, integration tests, and end-to-end tests must cover at least 90% of the codebase to ensure the system is bug-free and maintains expected performance.

Portability

- **Cross-Platform Compatibility:** The app must be compatible with most modern android The user experience should be consistent across all platforms.

- **Cloud Deployment:** The system should be deployable to popular cloud platforms like AWS or Azure, with minimal configuration changes required to switch between different cloud providers.

Scalability

- **Elastic Infrastructure:** The system infrastructure should automatically scale to handle increased loads during auction periods, adjusting to the number of active users and auction lots being uploaded.
- **Database Scalability:** The database must support horizontal scaling to accommodate future growth in auction lots, user registrations, and images stored.

Compliance/ Internationalization

- **POPIA Compliance:** The system must comply with South Africa's Protection of Personal Information Act (POPIA), especially concerning the collection, storage, and sharing of user data (i.e., Potential Bidders and Auctioneers).
- **Multi language support :** The app should support multiple languages, initially English and Afrikaans
- **GDPR:** If the system expands to handle international bidders, it should also comply with GDPR regulations to protect user data in the European Union.

Extensibility/ Interoperability

- **Plugin System:** The app should support the integration of external plugins (for analytics, marketing, etc.) to allow for custom functionality without modifying the core system.
- **RESTful API:** The system must provide a well-documented RESTful API to allow third-party applications (such as auction listing websites, payment gateways, and social media platforms) to interact with the app's core functionality.
 - The API should allow for creating, updating, and retrieving auction lots, along with user authentication and image uploads.
 - It should support JSON as the standard data interchange format to ensure compatibility with other systems.
- **Facebook & Instagram Integration:** The app must allow administrators to publish auction lot details directly to social media platforms (e.g., Facebook, Instagram) via API integration.

- The system must automate the creation of posts with auction images, descriptions, and links to the catalogue, streamlining the auction promotion process.
- Ensure the API tokens used to interact with these platforms are securely stored and regularly rotated to mitigate security risks.

4. Analysis Artifacts

4.1 Domain Modelling

Our domain model consists of the following bounded contexts:

1. User Management Context

- Entities: User, Role, Permission
- Responsible for user authentication, authorization, and profile management

2. Auction Lot Context

- Entities: Lot, Photo, Category
- Handles the creation, storage, and management of auction lots and associated photos

3. Catalogue Context

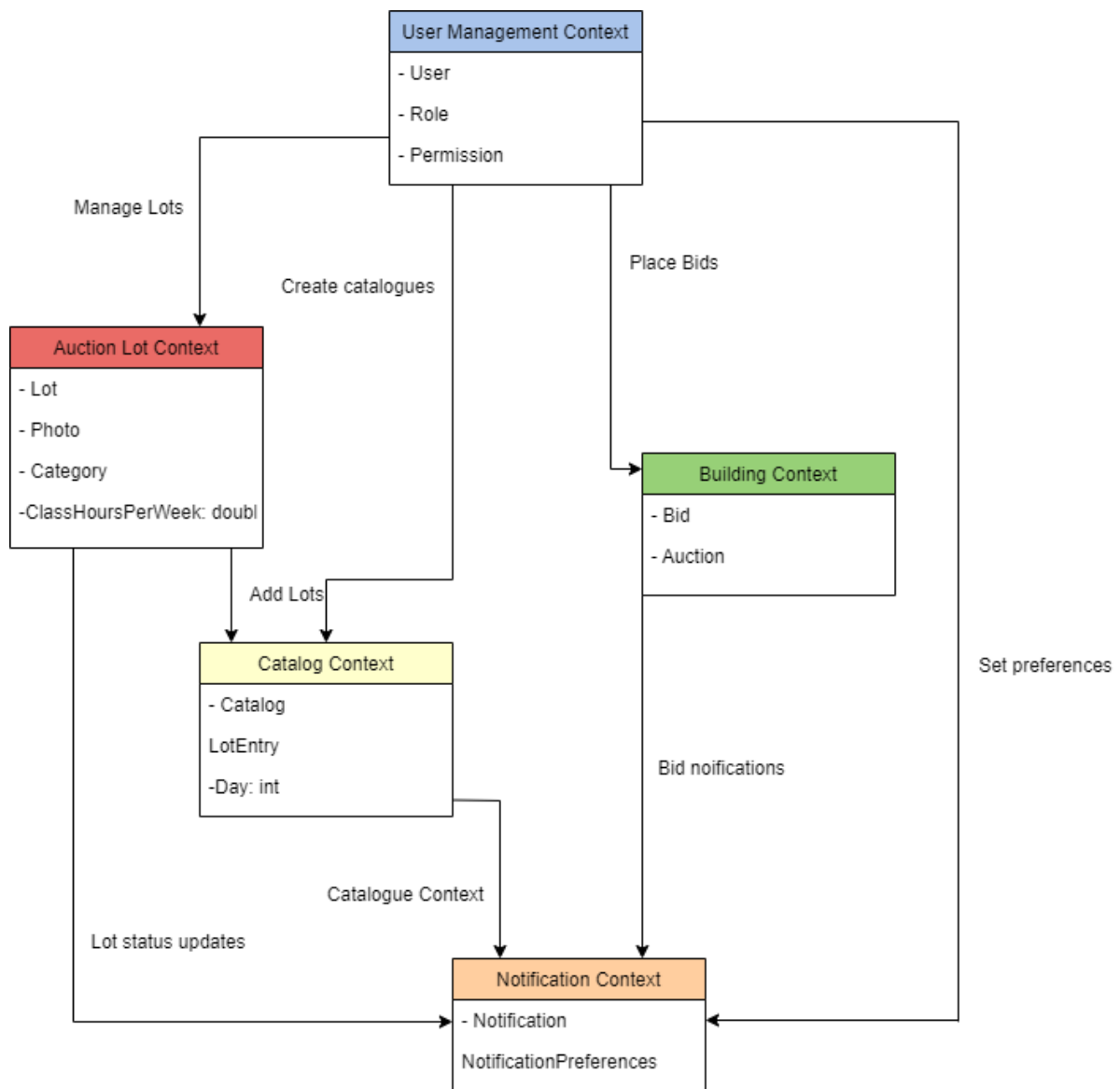
- Entities: Catalogue, Lot Entry
- Manages the organization of lots into catalogues and the generation of PDF documents

4. Bidding Context

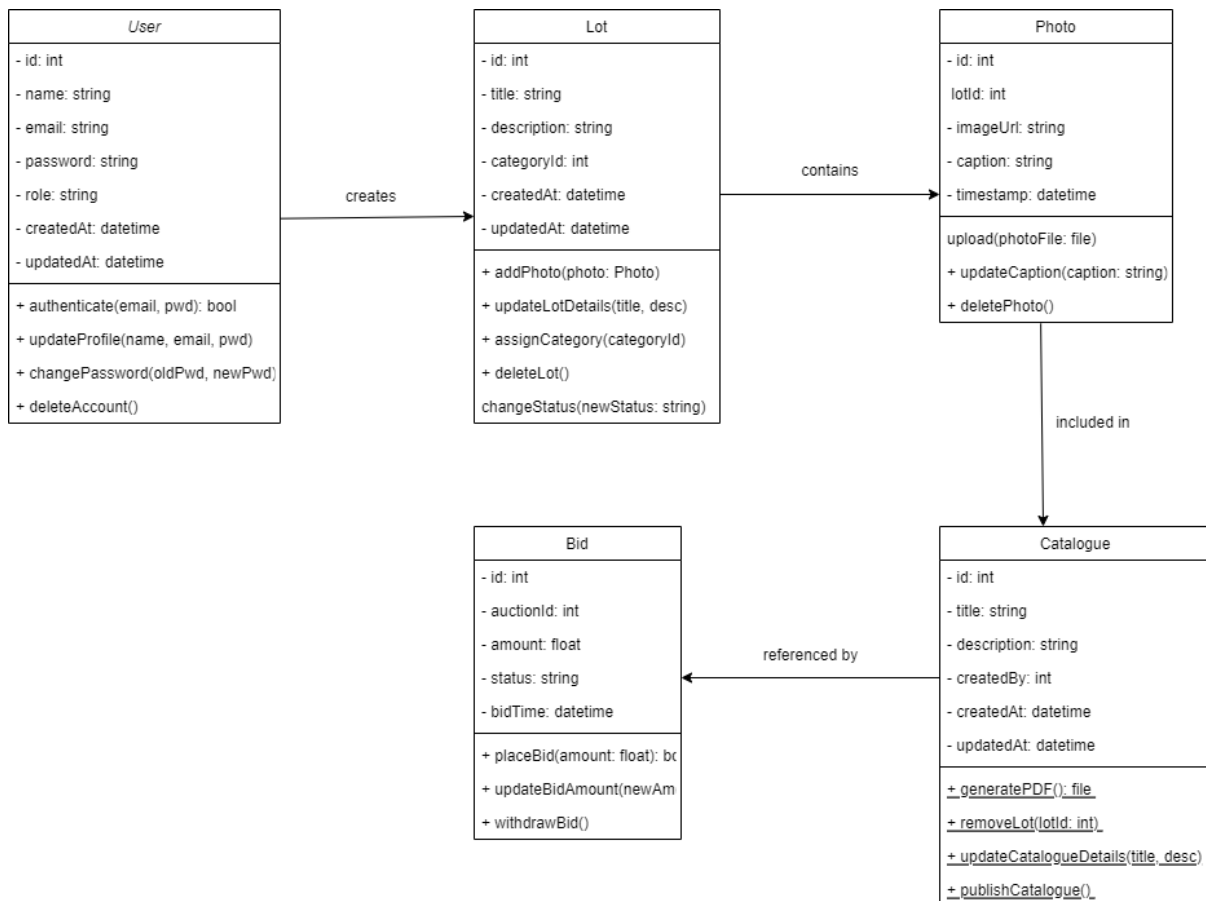
- Entities: Bid, Auction
- Handles the auction process and bid management (for future expansion)

5. Notification Context

- Entities: Notification, Notification Preference
- Manages system notifications and integration with external platforms



5. Design Artifacts



UML Diagram

6. Implementation Documentation

6.1 Lot Creation Flow

The following sequence diagram illustrates the critical flow of creating a new auction lot:

mermaid

Copy

```

sequenceDiagram
    participant Mobile App
    participant API Gateway
    participant Lambda
    participant S3
    participant RDS

    Mobile App->>API Gateway: POST /lots (with images)
    API Gateway->>Lambda: Trigger processLot function
    Lambda->>S3: Upload processed images
    S3-->>Lambda: Upload confirmation
    Lambda->>RDS: Create lot record
    RDS-->>Lambda: Lot created
    Lambda->>RDS: Store image metadata
  
```

```

RDS-->>Lambda: Image records created
Lambda-->>API Gateway: Lot creation success
API Gateway-->>Mobile App: 201 Created (Lot Details)

```

6.2 Critical Implementations

6.2.1 Image Processing Service

kotlin

```

class ImageProcessor {
    suspend fun processImages(images: List<Image>, lotId: String): List<ProcessedImage> {
        return coroutineScope {
            images.map { image ->
                async {
                    processImage(image, lotId)
                }
            }.awaitAll()
        }
    }

    private suspend fun processImage(image: Image, lotId: String): ProcessedImage {
        return withContext(Dispatchers.IO) {
            // Resize image
            val resized = image.scaleToSize(MAX_WIDTH, MAX_HEIGHT)

            // Compress image
            val compressed = resized.compressToQuality(COMPRESSION_QUALITY)

            // Generate unique filename
            val filename = "${lotId}_${UUID.randomUUID()}.jpg"

            ProcessedImage(
                filename = filename,
                data = compressed,
                size = compressed.size,
                aspectRatio = "${resized.width}:${resized.height}"
            )
        }
    }

    companion object {
        const val MAX_WIDTH = 1920
        const val MAX_HEIGHT = 1080
        const val COMPRESSION_QUALITY = 85
    }
}

```

6.2.2 PDF Catalog Generation Service

mermaid

```

sequenceDiagram
    participant Admin
    participant Lambda
    participant RDS
    participant S3

```

participant PDF Service

```

Admin-->>Lambda: Request Catalog Generation
Lambda-->>RDS: Fetch Auction Details
RDS-->>Lambda: Auction Data
Lambda-->>S3: Fetch Lot Images
S3-->>Lambda: Images
Lambda-->>PDF Service: Generate Catalog
PDF Service-->>S3: Store Generated PDF
S3-->>PDF Service: PDF URL
PDF Service-->>Lambda: Catalog Details
Lambda-->>Admin: Catalog Ready

```

6.2.3 Mobile App Navigation Implementation

kotlin

```

@HiltViewModel
class MainViewModel @Inject constructor(
    private val lotRepository: LotRepository,
    private val authManager: AuthManager
) : ViewModel() {

    private val _navigationState = MutableStateFlow<NavigationState>(NavigationState.Idle)
    val navigationState: StateFlow<NavigationState> = _navigationState.asStateFlow()

    fun navigateToLotCreation() {
        viewModelScope.launch {
            if (authManager.hasPermission(Permission.CREATE_LOT)) {
                _navigationState.value = NavigationState.LotCreation
            }
        }
    }

    sealed class NavigationState {
        object Idle : NavigationState()
        object LotCreation : NavigationState()
        data class LotDetail(val lotId: String) : NavigationState()
        object AuctionList : NavigationState()
        data class Error(val message: String) : NavigationState()
    }
}

```

6.3 Core Data Models

6.3.1 Lot Data Transfer Object (DTO)

kotlin

```

@Serializable
data class LotDto(
    val id: String,
    val auctionId: String,
    val lotNumber: String,
    val title: String,
    val description: String?,
    val images: List<ImageDto>,

```

```

    val status: LotStatus,
    val createdAt: Long,
    val updatedAt: Long
)

@Serializable
data class ImageDto(
    val id: String,
    val url: String,
    val sequenceNumber: Int,
    val aspectRatio: String
)

enum class LotStatus {
    DRAFT,
    READY,
    PUBLISHED,
    SOLD,
    WITHDRAWN
}

```

6.4 Repository Implementation

kotlin

```

class LotRepositoryImpl @Inject constructor(
    private val api: LotApi,
    private val lotDao: LotDao,
    private val imageProcessor: ImageProcessor,
    private val dispatchers: CoroutineDispatchers
) : LotRepository {

    override suspend fun createLot(lot: LotDto, images: List<Image>): Result<LotDto> =
        withContext(dispatchers.io) {
            try {
                // Process images
                val processedImages = imageProcessor.processImages(images, lot.id)

                // Upload to API
                val uploadedLot = api.createLot(lot.copy(
                    images = processedImages.map { it.toImageDto() }
                ))

                // Cache locally
                lotDao.insertLot(uploadedLot.toLotEntity())

                Result.success(uploadedLot)
            } catch (e: Exception) {
                Result.failure(e)
            }
        }

    override fun observeLots(auctionId: String): Flow<List<LotDto>> =
        lotDao.observeLots(auctionId)
}

```



```

        .map { lots -> lots.map { it.toLotDto() } }
        .distinctUntilChanged()
        .flowOn(dispatchers.io)
    }

```

6.5 Error Handling Implementation

kotlin

```

sealed class ApiResult<out T> {
    data class Success<T>(val data: T) : ApiResult<T>()
    data class Error(val code: Int, val message: String) : ApiResult<Nothing>()
    data class Exception(val e: Throwable) : ApiResult<Nothing>()
}

suspend fun <T> safeApiCall(dispatcher: CoroutineDispatcher, apiCall: suspend () -> T): ApiResult<T>
=
    withContext(dispatcher) {
        try {
            ApiResult.Success(apiCall())
        } catch (e: HttpException) {
            ApiResult.Error(e.code(), e.message())
        } catch (e: Exception) {
            ApiResult.Exception(e)
        }
    }
}

```

6.6 Security Implementation

kotlin

```

@Singleton
class SecurityManager @Inject constructor(
    private val keyStore: KeyStore,
    private val encryptionService: EncryptionService
) {
    fun secureData(data: String): String {
        val key = keyStore.getKey(KEY_ALIAS, null)
        return encryptionService.encrypt(data, key)
    }

    fun verifyData(data: String, signature: String): Boolean {
        val key = keyStore.getKey(KEY_ALIAS, null)
        return encryptionService.verify(data, signature, key)
    }

    companion object {
        private const val KEY_ALIAS = "RIHLAZANA_KEY"
    }
}

```

6.7 Testing Implementation

kotlin

```

@RunWith(JUnit4::class)

```

```
class LotRepositoryTest {
    @get:Rule
    val coroutineRule = MainCoroutineRule()

    private lateinit var repository: LotRepository
    private lateinit var api: FakeLotApi
    private lateinit var dao: FakeLotDao
    private lateinit var imageProcessor: FakeImageProcessor

    @Before
    fun setup() {
        api = FakeLotApi()
        dao = FakeLotDao()
        imageProcessor = FakeImageProcessor()
        repository = LotRepositoryImpl(
            api = api,
            lotDao = dao,
            imageProcessor = imageProcessor,
            dispatchers = coroutineRule.testDispatcherProvider
        )
    }

    @Test
    fun `createLot processes images and uploads successfully`() = runTest {
        // Given
        val lot = TestData.createTestLot()
        val images = TestData.createTestImages()

        // When
        val result = repository.createLot(lot, images)

        // Then
        assertThat(result.isSuccess).isTrue()
        assertThat(api.uploadedLots).hasSize(1)
        assertThat(dao.cachedLots).hasSize(1)
    }
}
```

This offers a comprehensive overview of the key technical implementations in the Rihlazana Auctions mobile application. The code examples demonstrate best practices in:

1. Asynchronous programming with Kotlin Coroutines
2. Clean Architecture principles
3. Dependency injection
4. Error handling
5. Security implementation
6. Unit testing
7. Repository pattern implementation

7. Data Schema Documentation

Overview

The Rihlazana Auctions application uses Amazon RDS (PostgreSQL) for structured data storage and Amazon S3 for image storage. This hybrid approach allows us to maintain relational integrity for auction data while efficiently handling large media files.

Entity Relationship Diagram (ERD)

```
erDiagram
    User ||--o{ Lot : creates
    User {
        int user_id PK
        string email
        string password_hash
        string full_name
        string phone_number
        enum role
        timestamp created_at
        timestamp updated_at
    }
    Lot ||--o{ Image : contains
    Lot {
        int lot_id PK
        int creator_id FK
        int auction_id FK
        string lot_number
        string title
        text description
        decimal reserve_price
        enum status
        timestamp created_at
        timestamp updated_at
    }
    Image {
        int image_id PK
        int lot_id FK
        string s3_url
        int sequence_number
        string aspect_ratio
        int file_size
        timestamp uploaded_at
    }
    Auction ||--o{ Lot : contains
    Auction {
        int auction_id PK
        string title
        date auction_date
        enum status
    }
```

```

    text location
    timestamp created_at
    timestamp updated_at
  }
  Catalog ||--o{ Lot : includes
  Catalog {
    int catalog_id PK
    int auction_id FK
    string pdf_url
    timestamp generated_at
    string version
  }

```

Database Tables Description

1. User Table

Stores user information and authentication details.

```

CREATE TABLE users (
  user_id SERIAL PRIMARY KEY,
  email VARCHAR(255) UNIQUE NOT NULL,
  password_hash VARCHAR(255) NOT NULL,
  full_name VARCHAR(255) NOT NULL,
  phone_number VARCHAR(20),
  role VARCHAR(20) NOT NULL CHECK (role IN ('ADMIN', 'OPERATIONS', 'BIDDER')),
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

```

2. Auction Table

Represents individual auction events.

```

CREATE TABLE auctions (
  auction_id SERIAL PRIMARY KEY,
  title VARCHAR(255) NOT NULL,
  auction_date DATE NOT NULL,
  status VARCHAR(20) NOT NULL CHECK (status IN ('DRAFT', 'SCHEDULED', 'ACTIVE', 'COMPLETED', 'CANCELLED')),
  location TEXT,
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

```

3. Lot Table

Stores information about individual auction lots.

```

CREATE TABLE lots (
  lot_id SERIAL PRIMARY KEY,

```

```

creator_id INTEGER REFERENCES users(user_id),
auction_id INTEGER REFERENCES auctions(auction_id),
lot_number VARCHAR(50) NOT NULL,
title VARCHAR(255) NOT NULL,
description TEXT,
reserve_price DECIMAL(10,2),
status VARCHAR(20) NOT NULL CHECK (status IN ('DRAFT', 'READY', 'PUBLISHED', 'SOLD',
'WITHDRAWN')),
created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
UNIQUE(auction_id, lot_number)
);

```

4. Image Table

Manages metadata for lot images stored in S3.

```

CREATE TABLE images (
  image_id SERIAL PRIMARY KEY,
  lot_id INTEGER REFERENCES lots(lot_id),
  s3_url VARCHAR(512) NOT NULL,
  sequence_number INTEGER NOT NULL,
  aspect_ratio VARCHAR(20),
  file_size INTEGER,
  uploaded_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  UNIQUE(lot_id, sequence_number)
);

```

5. Catalog Table

Tracks generated PDF catalogs.

```

CREATE TABLE catalogs (
  catalog_id SERIAL PRIMARY KEY,
  auction_id INTEGER REFERENCES auctions(auction_id),
  pdf_url VARCHAR(512) NOT NULL,
  generated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  version VARCHAR(20) NOT NULL,
  UNIQUE(auction_id, version)
);

```

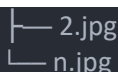
Image Storage in Amazon S3

Images are stored in S3 using the following path structure:

```

s3://rihlazana-auctions/
├─ lots/
│   └─ {auction_id}/
│       └─ {lot_number}/
│           └─ 1.jpg

```



```
├── 2.jpg
└── n.jpg
```

Key Design Decisions

1. Relational Database Choice

- PostgreSQL was chosen for its robust ACID compliance and support for complex queries
- The relational model ensures data integrity across auctions, lots, and images
- Built-in constraint support helps maintain data quality

2. Image Storage Strategy

- Images stored in S3 rather than the database for:
 - Better performance
 - Cost efficiency
 - Easier CDN integration
 - Simplified backup management

3. Indexing Strategy

```
-- Frequently accessed queries
CREATE INDEX idx_lots_auction_id ON lots(auction_id);
CREATE INDEX idx_images_lot_id ON images(lot_id);
CREATE INDEX idx_lots_status ON lots(status);
CREATE INDEX idx_auctions_date ON auctions(auction_date);
```

4. Constraints and Data Integrity

- Foreign key constraints ensure referential integrity
- CHECK constraints enforce valid status values
- UNIQUE constraints prevent duplicate lot numbers within auctions
- Default timestamps track record creation and updates

Performance Considerations

1. Partitioning

- Tables will be partitioned by auctiondate for efficient historical data management
- Improves query performance for active auctions

2. Caching Strategy

- Frequently accessed lot data cached in application layer
- Image metadata cached to reduce database load
- S3 integrated with CloudFront for efficient image delivery

8. Architecture Artifacts

8.1 Design Patterns

Our application will utilize the following design patterns:

1. Repository Pattern: For abstracting data access logic.
2. Factory Pattern: For creating complex objects, particularly in the photo processing pipeline.
3. Observer Pattern: For implementing a publish-subscribe model for real-time updates.
4. Singleton Pattern: For managing shared resources like database connections.

8.2 Architecture Patterns

We will be using a Clean Architecture approach with the following layers:

1. Presentation Layer (UI)
2. Use Case Layer (Application Logic)
3. Domain Layer (Business Logic)
4. Data Layer (Data Access)

This architecture allows for separation of concerns and makes the system more testable and maintainable.

8.3 Cloud

We will be using Azure for our cloud infrastructure. Our cloud architecture includes:

1. Azure App Service for hosting the backend API
2. Azure Blob Storage for storing photos and documents
3. Azure SQL Database for relational data storage
4. Azure Functions for serverless processing of photos and PDF generation
5. Azure API Management for managing our APIs
6. Azure Active Directory for identity management

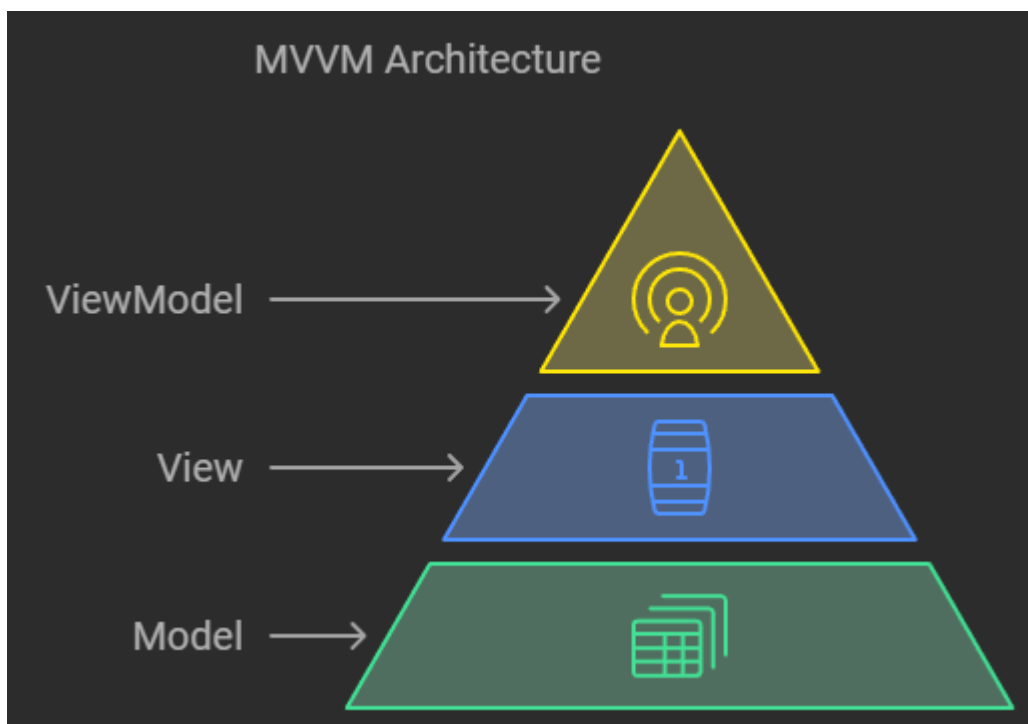
Architecture artifacts provide a comprehensive overview of the design decisions, patterns, and technologies employed in developing the auction cataloguing application. These artifacts ensure the system's scalability, maintainability, and reliability, while aligning with the overall architectural strategy.

Design and Architecture Patterns

Design Patterns

1. Model-View-View Model (MVVM) Architecture

- **Justification:** Selected to decouple business logic from UI code, enhancing testability, maintainability, and scalability.
- **Details:**
 - **Model:** Represents the data layer, managing data retrieval from APIs and local databases.
 - **View:** Displays UI components, such as auction lots, images, and the PDF catalogue.
 - **View Model:** Serves as the intermediary, binding data to the View and handling user interactions.
- **Advantages:**
 - Promotes separation of concerns, facilitating straightforward testing.
 - Supports reactive programming through Kotlin's coroutines and Live Data for asynchronous data handling.
 -



2. Repository Pattern

- **Justification:** Abstracts data access logic and centralizes operations for data retrieval, storage, and management.
- **Details:**

- Manages interactions between the View Model and data sources (remote API and local database).
- Simplifies testing by allowing easy mocking of data sources and simplifies swapping sources when needed.

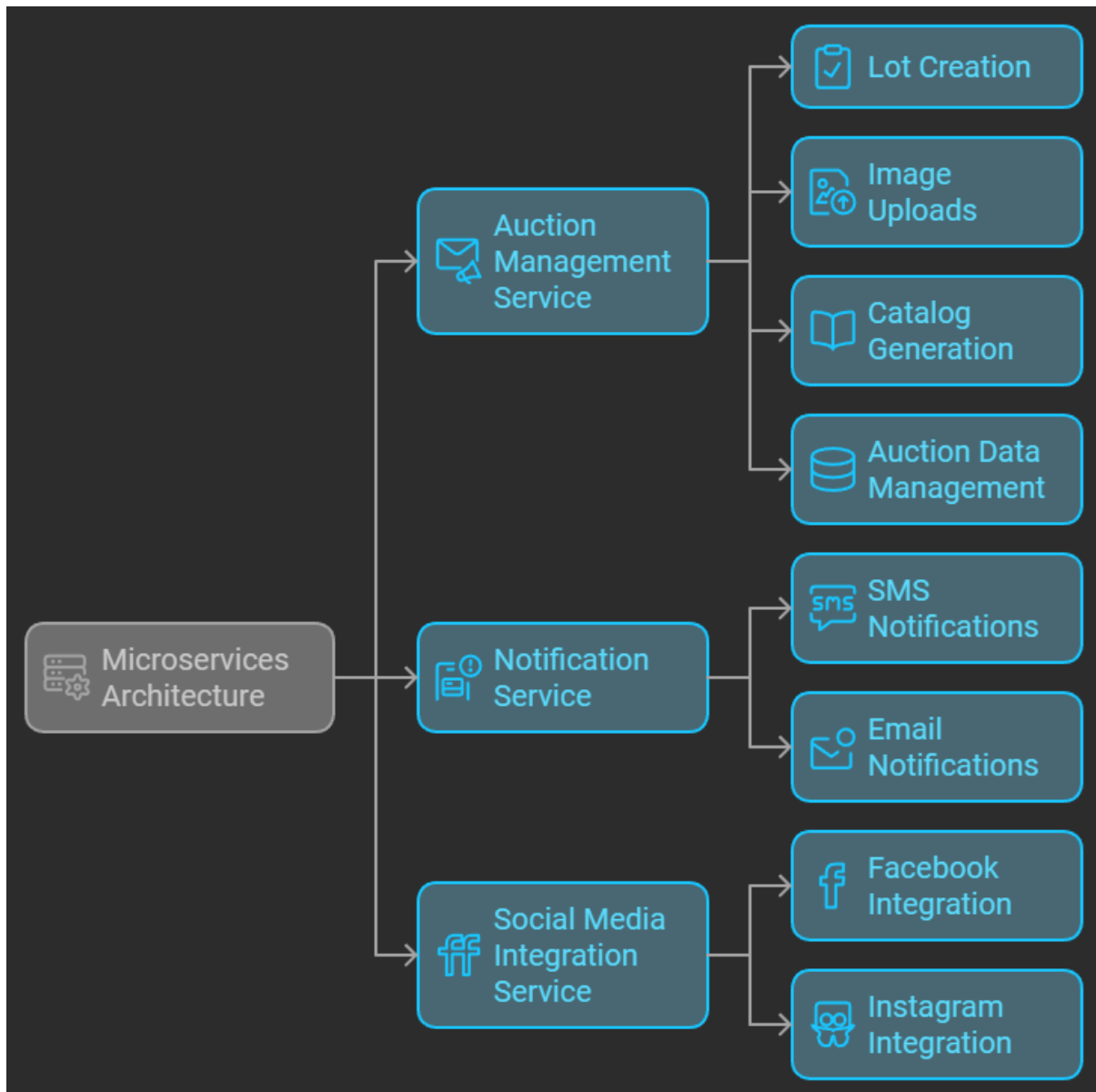
Architecture Patterns

1. Client-Server Architecture

- **Justification:** Separates the user interface from backend logic for improved organization and scalability.
- **Details:**
 - **Client (Mobile Application):** Handles user interactions, including capturing photos, creating lots, viewing auctions, and exporting catalogues.
 - **Server:** Manages auction data, stores image, generates catalogues, and integrates with external services.
- **Advantages:**
 - **Scalability:** Capable of handling multiple client requests concurrently.
 - **Maintainability:** Supports independent updates and maintenance.

2. Microservices Architecture (for API)

- **Justification:** Divides backend functionality into smaller, independent services, allowing for scalability and flexibility.
- **Details:**
 - **Auction Management Service:** Manages lot creation, image uploads, catalogue generation, and auction data.
 - **Notification Service:** Sends notifications to users and bidders.
 - **Social Media Integration Service:** Handles integration with platforms like Facebook and Instagram for promotional activities.
- **Advantages:**
 - Enables independent deployment and scaling of services.
 - Facilitates separate development and testing efforts.



Stacks and Queues

- **Usage:** Standard data structures utilized for task scheduling and managing user interactions.
- **Example:**
 - **Queue:** Processes and uploads images in batches (lots) to maintain FIFO (First In, First Out) order, preventing overlapping uploads and bottlenecks.
 - **Stack:** Manages user interactions and navigation, enabling functionalities such as "undo" operations and maintaining a history of user actions.

Linked Lists

- **Usage:** Employed for managing collections of auction lots or user actions that require dynamic resizing.
- **Example:**

- A linked list facilitates efficient insertion and deletion of lots, enabling dynamic updates as lots are added or removed.

Code Quality and Documentation

- Adheres to clean code principles, ensuring a modular and well-structured codebase. Comprehensive documentation includes:
 - **Data Classes and DTOs:** Structured data transfer objects (DTOs) for standardized communication between microservices.
 - **Code Quality:** Follows best practices for readability, modularity, and maintainability, with detailed comments and a guiding README.md file.

Communication and Integration

1. Frontend and Backend Communication

- **Web Services and APIs:** RESTful services facilitate communication between the front-end and back-end using standard HTTP protocols.
- **Integration Layer:** Ensures consistent data flow between services, with the repository pattern abstracting data access for better maintainability.

2. Third-Party API Integration

- **Social Media Integration:** Promotes auction events through APIs, automatically posting auction details on social media platforms.
- **Geolocation Services:** Offers functionalities like tracking auction lots and directing users to auction sites via third-party APIs.

3. Security for API Integrations

- Utilizes OAuth 2.0 for secure authentication with third-party APIs and implements rate limiting to comply with API terms of service.

Cloud Architecture

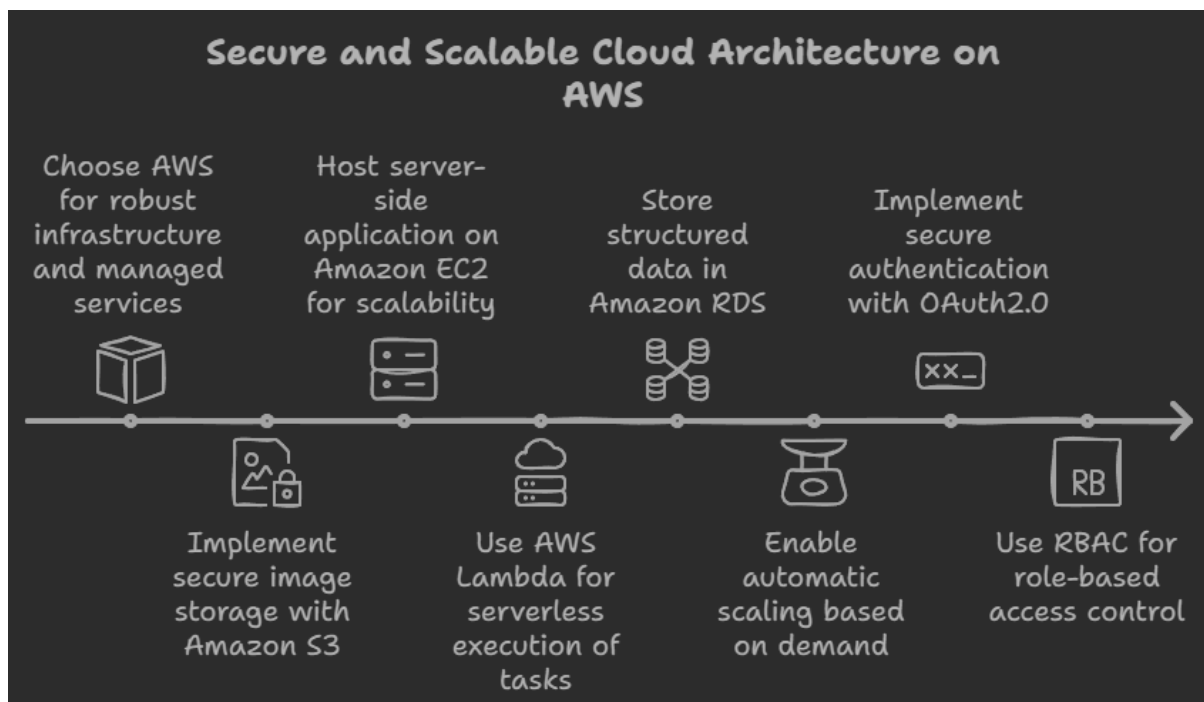
1. Cloud Deployment on AWS

- **Justification:** Selected for its robust, scalable infrastructure and extensive range of managed services.
- **Details:**
 - **Amazon S3:** Secure image storage accessible by the admin team.
 - **Amazon EC2:** Hosts server-side applications to ensure scalability.
 - **AWS Lambda:** Executes lightweight tasks such as catalogue generation and barcode scanning.
 - **Amazon RDS:** Stores structured auction data and user information.
- **Advantages:**
 - **Scalability:** Automatically adjusts resources based on demand.

- **High Availability:** Ensures system accessibility through a distributed architecture.

2. Backup and Disaster Recovery

- **Justification:** Ensures data protection and recovery in unforeseen failures or disasters.
- **Details:**
 - Automated backups of databases and image storage.
 - Multi-AZ deployments for critical components to ensure high availability.



9. Security

Potential Threat Actors

- **Internal Threats:** Unauthorized users within the organization such as a disgruntled employee, or an unauthorized member of the operations team could attempt to upload malicious content or modify auction details.
- **External Threats:** Hackers, unauthorized external users, or automated bots attempting to exploit weaknesses in the login system, APIs, or data storage to steal or modify sensitive data (e.g., auction information, images).
- **Competitors:** May attempt to gain unauthorized access to auction details or manipulate public data.

Potential Threat Vectors

- **Phishing Attacks:** Users could be tricked into revealing their login credentials through phishing attempts, which attackers could then use to gain unauthorized access.
- **API Vulnerabilities:** The application's API endpoints could be attacked through injection attacks (SQL/NoSQL), cross-site scripting (XSS), or man-in-the-middle (MITM) attacks.
- **Unauthorized Data Access:** Weak encryption or poor access control could allow attackers to access stored images, auction details, or bidder data.

Mitigations for Threats

- **Phishing Prevention:** Implement **multi-factor authentication (MFA)** to mitigate the risk of stolen credentials through phishing. User education can be enforced to recognize phishing attempts.
- **API Security:** The use of **input validation** to prevent injection attacks, and **rate limiting** to reduce the risk of denial-of-service (DoS) attacks on API endpoints.
- **Data Access Control:** Strong encryption at rest and in transit (AES-256, TLS) ensures that unauthorized data access is mitigated. **Role-Based Access Control (RBAC)** ensures only authorized personnel access sensitive data.
- **Session Management:** Implement **JWT expiration** and token refreshing to minimize the risk of session hijacking and stolen tokens being used maliciously.

Economy of Mechanism

The principle of **economy of mechanism** was considered to avoid unnecessary complexity in security mechanisms:

- **OAuth 2.0** simplifies authentication by offloading it to trusted third-party services (Google, Facebook, etc.), which reduces the need for complex custom login systems.
- **IAM Policies** in AWS ensure minimalistic access control, meaning that only required permissions are granted without adding unnecessary security layers, reducing the potential for misconfiguration.

Balancing Security and Usability

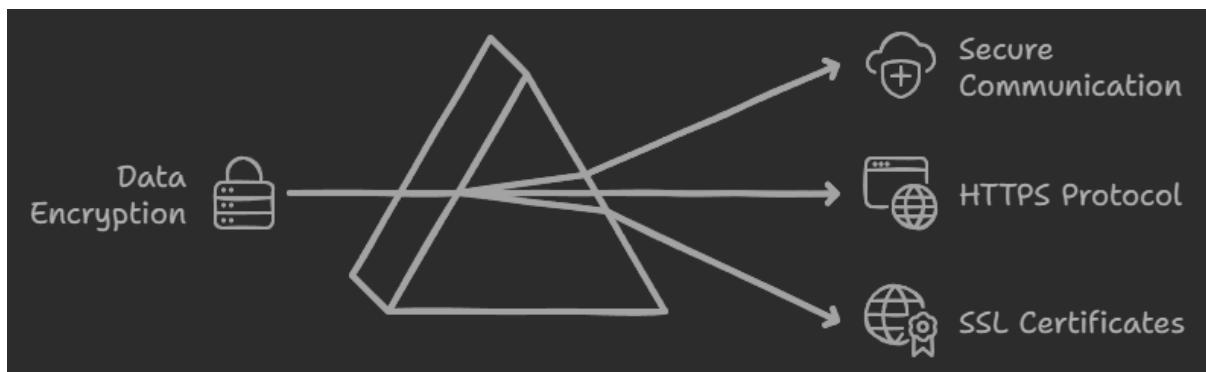
- The application strikes a balance by implementing security mechanisms like **social login** and **OAuth 2.0**, which make it easy for users to log in securely without compromising the protection of sensitive data.
- **User-friendly encryption:** Data is encrypted without requiring end-users to manage encryption keys or manually deal with certificates, which ensures security without impacting usability.
- **RBAC** ensures that while critical actions are protected, regular users (such as potential bidders) still have seamless access to view auctions without being burdened by unnecessary security steps.

Security Consideration for Every Data Object

- Every sensitive data object in the system (e.g., auction lots, user information, and auction images) is protected by **RBAC**, meaning that only users with proper permissions can access or modify this data.
- **Complete mediation** is achieved by enforcing consistent access controls at each layer (from UI to API to database), ensuring that unauthorized users can't bypass permissions to access sensitive data.

Complete Mediation

- **Complete mediation** is ensured using centralized authentication and authorization mechanisms (OAuth 2.0, JWT), where every request must pass through the appropriate security checks before access to any data object is granted.
- Access to all backend resources (e.g., AWS S3, RDS) is tightly controlled using **IAM roles** and **VPCs** to ensure that even legitimate users cannot access data they are not authorized to view or modify.



10. DevOps

10.1 GitHub Actions Pipeline

Our CI/CD pipeline using GitHub Actions includes the following steps:

1. Code Checkout
2. Environment Setup
3. Dependency Installation
4. Linting
5. Unit Testing
6. Integration Testing
7. Build
8. Security Scan
9. Deployment to Staging
10. Smoke Tests
11. Deployment to Production (manual trigger)

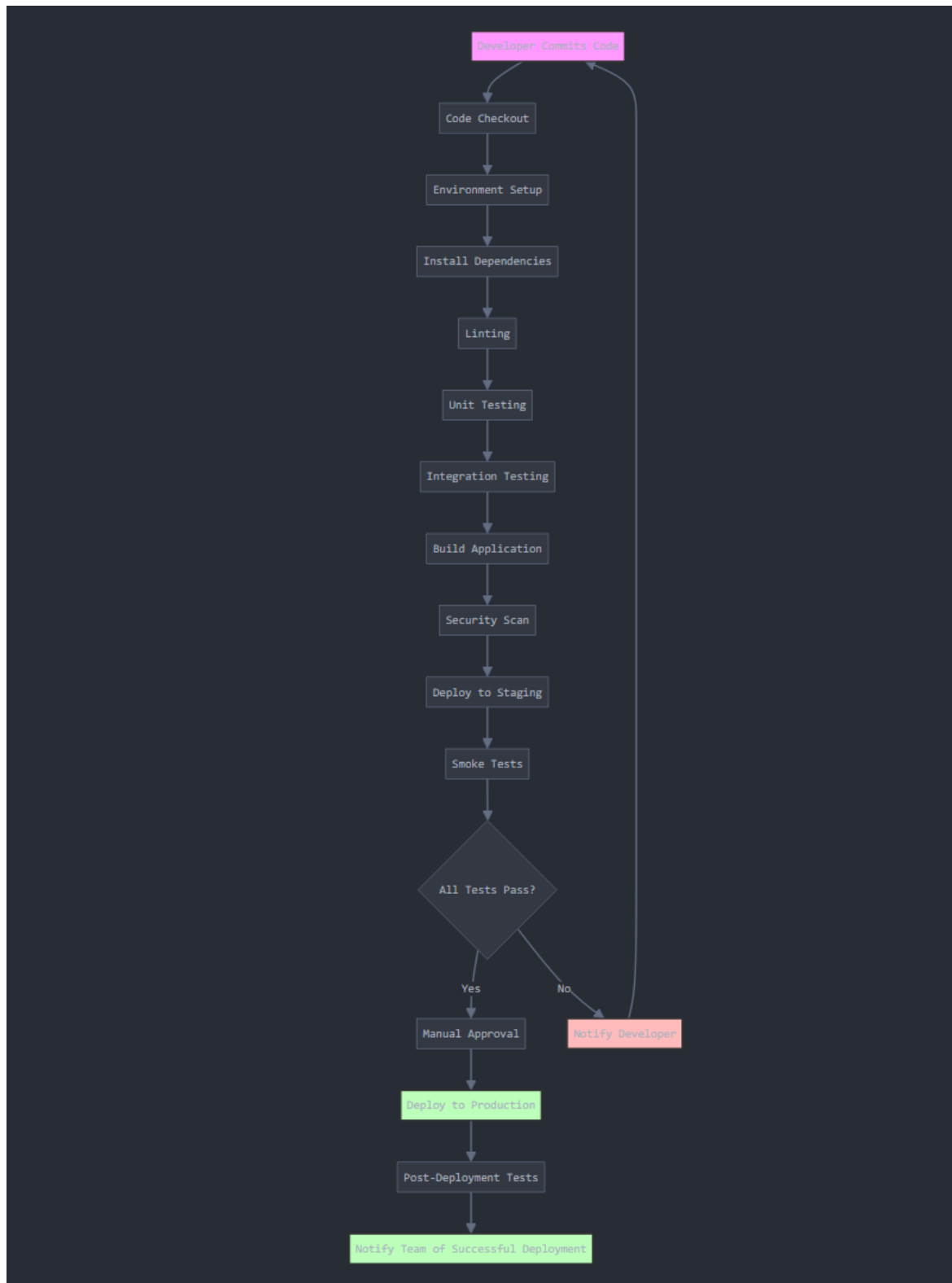


Diagram of the GitHub Actions workflow

11. Running Costs

○ 11.1 Predicted User Growth

- We anticipate the following user growth for the Rihlazana Auctions mobile app over the next two years:
 - Initial users at launch: 200
 - End of Year 1: 5,000 users
 - End of Year 2: 10,000 users
- This growth is based on Rihlazana Auctions' current customer base and projected market expansion.

○ 11.2 Scaling Points for AWS Services

- Amazon EC2 (t3.small):
 - Scales at 3,000 users
 - Cost: R800/month per instance
 - New instance added for every additional 3,000 users
- Amazon S3:
 - Scales with data volume, not directly with user count
 - Estimated 5MB per user
 - Cost: R0.30 per GB/month
- Amazon RDS (db.t3.small):
 - Scales at 8,000 users
 - Cost: R1,000/month for single instance
 - Upgrade to multi-AZ deployment at 8,000 users: R2,000/month
- AWS Lambda:
 - Scales automatically with usage
 - Cost: R0.20 per million invocations
- Amazon API Gateway:
 - Scales automatically
 - Cost: R4 per million API calls

○ 11.3 Predictive Models (Monthly Basis over Two Years)

○ Best Case Scenario (High Growth)

- Year 1 End: 7,500 users
- Year 2 End: 15,000 users

○ Worst Case Scenario (Low Growth)

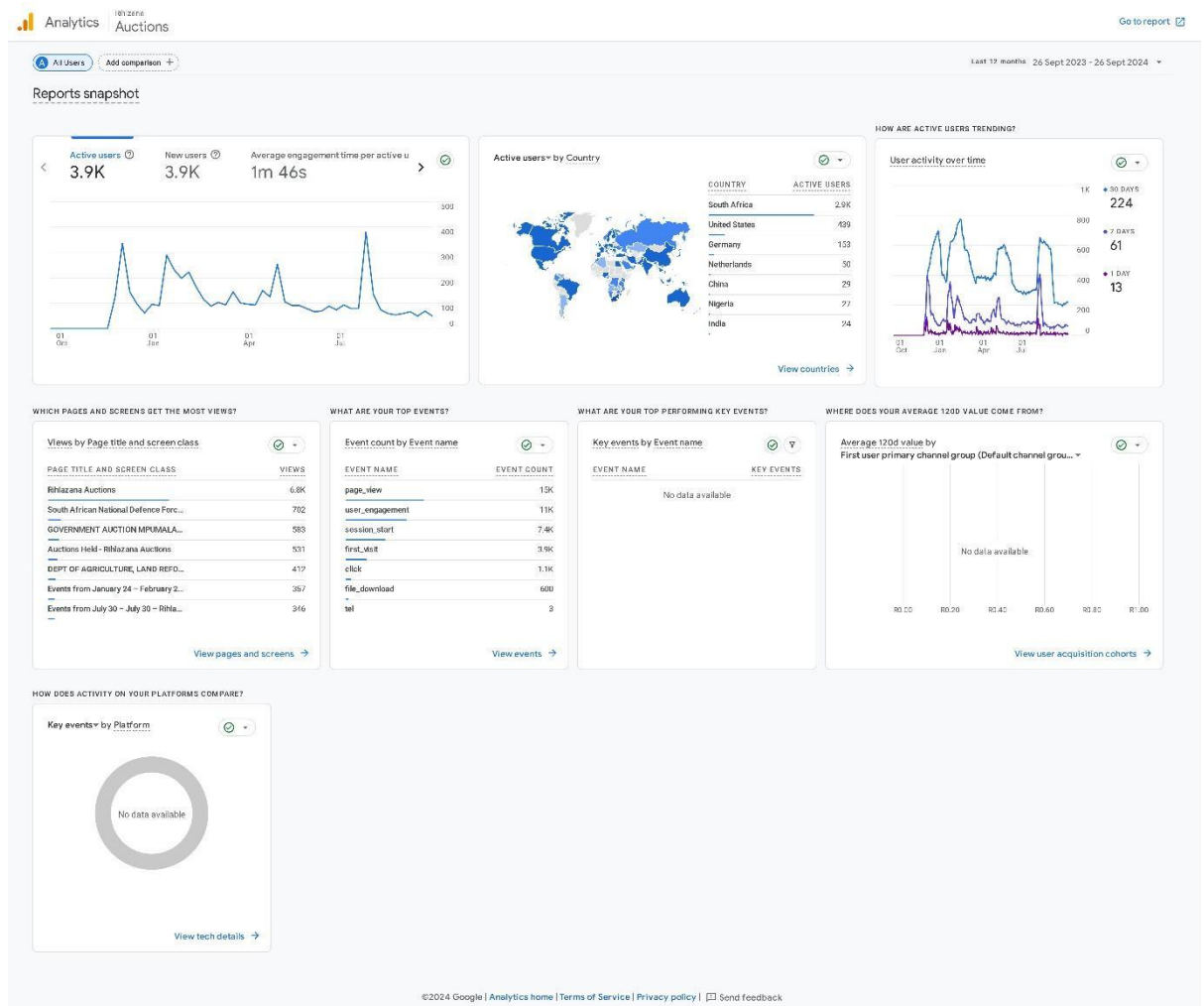
- Year 1 End: 2,500 users
- Year 2 End: 5,000 users

- Average Growth Scenario

- Year 1 End: 5,000 users
- Year 2 End: 10,000 users
- [A table or graph would be inserted here showing the monthly user growth and associated costs for each scenario over the two-year period.]

- 11.4 Technology Adjustments at Scale

- Given the adjusted user growth projections, major technology replacements aren't necessary within the first two years. However, we anticipate some optimizations:
- Database Enhancement:
 - Current: Amazon RDS (Single AZ)
 - Scaling Point: 8,000 users
 - Adjustment: Upgrade to multi-AZ deployment
 - Reason: Improved availability and performance for growing user base
 - Cost Comparison:
 - Single AZ (up to 8,000 users): R1,000/month
 - Multi-AZ (8,000+ users): R2,000/month
- Compute Optimization:
 - Current: Amazon EC2 (t3.small)
 - Scaling Point: No immediate limit within projected growth
 - Potential Adjustment: Use Auto Scaling groups
 - Reason: Automatically adjust capacity to maintain steady performance
 - Cost Impact: Potential savings of 15-20% on EC2 costs through optimized resource usage
- Serverless Expansion:
 - Current: AWS Lambda
 - Scaling Point: No immediate limit within projected growth
 - Potential Adjustment: Migrate more functionality to Lambda
 - Reason: Improve scalability and potentially reduce costs for certain operations
 - Cost Impact: Potential reduction in EC2 costs, offset by increased Lambda usage



12. Change Management

Organizational Adoption

Rihlazana Auctions stands to gain significant competitive advantage by adopting our mobile application. By automating the cataloguing process, we'll dramatically reduce the time and resources currently expended on manual data entry and photo management. This efficiency will not only cut operational costs but also minimize errors, ensuring higher quality auction data. Furthermore, the app's analytics capabilities will provide valuable insights into bidder behaviour, enabling data-driven decision-making that can inform future business strategies.

To facilitate adoption, we propose a phased rollout approach. Beginning with a pilot program involving key stakeholders, we'll gather crucial feedback to refine the app before full deployment. This approach allows for gradual integration into existing workflows, minimizing disruption while

maximizing the potential for success. By demonstrating tangible improvements in efficiency and data quality during this pilot phase, we'll build a compelling case for organization-wide adoption.

User Adoption

For both staff and potential bidders, our app offers a user-friendly interface that simplifies the auction process. Staff members will appreciate the streamlined workflow that allows them to manage auctions on-the-go, freeing up time for other valuable tasks. Potential bidders will benefit from real-time updates, high-quality images, and easy navigation, enhancing their overall auction experience.

To drive user adoption, we'll implement a multi-faceted strategy. For staff, we'll provide comprehensive training programs and ongoing support to ensure they're comfortable with the new system. For bidders, we'll offer incentives for early adoption, such as exclusive early access to new listings. By continuously gathering and acting on user feedback, we'll demonstrate our commitment to meeting their needs, fostering a sense of ownership and engagement with the platform.

Adoption and Support Strategy

Our strategy to gain and maintain adoption centres on clear communication, robust support, and continuous improvement. We'll launch a targeted communication campaign highlighting the app's benefits for each user group. A dedicated support team will be available to address any issues promptly, ensuring a smooth transition and ongoing satisfaction.

To maintain momentum post-launch, we'll implement a regular update schedule, introducing new features and improvements based on user feedback. This approach not only keeps the app relevant and engaging but also demonstrates our ongoing commitment to enhancing the user experience.

Our maintenance and support strategy are designed for long-term sustainability. We'll establish a multi-tiered support system, from self-help resources to personalized assistance for complex issues. Regular security updates and performance optimizations will be scheduled to ensure the app remains secure and efficient. By proactively monitoring system health and user feedback, we can address potential issues before they impact the user experience.

References

Amazon Web Services. (2024). AWS Documentation. Retrieved from <https://docs.aws.amazon.com/>

Anderson, G. (2022). Become an Android Developer with Kotlin. Packt Publishing.

Android Developers. (2024). Android Developer Documentation. Retrieved from <https://developer.android.com/docs>

AWS Architecture Blog. (2024). "Best Practices for Building on AWS." Retrieved from <https://aws.amazon.com/blogs/architecture/>

Coplien, J., & Harrison, N. (2004). Organizational Patterns of Agile Software Development. Prentice Hall.

Google. (2024). Material Design Guidelines. Retrieved from <https://material.io/design>

JetBrains. (2024). Compose Multiplatform Documentation. Retrieved from <https://www.jetbrains.com/lp/compose-multiplatform/>

Khor, W. L. (2023). "Advanced Android Testing." Retrieved from <https://developer.android.com/training/testing>

Klepmann, M. (2017). Designing Data-Intensive Applications. O'Reilly Media.

Kotlin Foundation. (2024). Kotlin Documentation. Retrieved from <https://kotlinlang.org/docs/home.html>

The Independent Institute of Education. (2024). XBCAD7319/XBCAD7329 Work Integrated Learning Module Manual. The Independent Institute of Education (Pty) Ltd.

- *Primary reference for project requirements and assessment criteria*
- *Annexure A: Cognitive Overload - Reference for managing complexity in software development*
- *Annexure C: Recommended Tools - Reference for technology choices*
- *Annexure D: Scrum Agile Primer - Reference for development methodology*
- *Annexure F: Quick Start to WIL - Reference for project initialization*
- *Annexure G: PoE Rubric - Reference for portfolio requirements*