ChatGPT

System Requirements for Australian Conveyancer Marketplace Platform

Overview

This document outlines the system requirements for a two-sided conveyancing marketplace in Australia (a "Fiverr for conveyancers"). It covers both functional and non-functional requirements, based on a containerized microservice architecture (as defined in the Docker Compose setup) and regulatory guidelines. The platform enables property buyers/sellers to connect with licensed conveyancers, negotiate jobs via chat, exchange documents, and transact through milestone-based escrow payments. Key considerations include strict compliance with Australian regulations (licensing verification, trust account rules, KYC/AML, electronic transactions) and robust technical architecture for security, scalability, and observability. The following sections detail the required functionality, technical stack, microservices, infrastructure, security/compliance measures, data policies, performance targets, development environment, and integration points.

Functional Requirements

User Onboarding & Identity Verification

- **Multi-Role Registration:** Users can register as **customers (clients)** seeking conveyancing services or **conveyancers (service providers)**. The platform shall support account creation with email and phone verification (e.g. OTP or activation link) to ensure valid contact information.
- **Know-Your-Customer (KYC):** The system must enforce KYC checks on all users, with enhanced verification for conveyancers. Customers and conveyancers are required to submit government-issued photo ID and proof of address as part of onboarding. The platform integrates with a third-party identity verification service to validate documents and user identity per Australian AML/KYC requirements [1] [2]. Conveyancers must additionally upload their professional **licence details** (licence number, issuing state, expiry) and **professional indemnity insurance** certificate for verification. Any **KYC status** (e.g. pending, verified, failed) is recorded in the user profile [2]. Users who fail KYC or whose documents expire cannot transact until issues are resolved.
- **Two-Factor Authentication:** For security, the platform shall support optional 2FA (e.g. one-time SMS codes or authenticator app) upon login, **mandated for conveyancers** to protect their accounts [2]. This mitigates fraud risk in a high-stakes sector.

Conveyancer Profile Management

- **Profile Creation:** Verified conveyancers can create a public profile listing their credentials and services. Profile fields include licence number & state, licence expiry date, insurance provider & expiry, areas of expertise, service pricing (hourly or fixed-fee packages), years of experience, languages spoken, and a bio. The system will clearly display the states in which the conveyancer is licensed to operate.

- **Licence & Insurance Verification:** The platform must validate each conveyancer's licence against the relevant **state government register** and ensure it is current [3]. For example, the system might query or reference NSW Fair Trading's licence register or Consumer Affairs Victoria's public register to confirm the licence number, status, and expiry [3]. Likewise, the conveyancer's professional indemnity insurance status must be recorded and kept up to date (in Victoria, a minimum AU$2M cover is required, and licences are suspended if insurance lapses [4]). The system should send reminders to conveyancers before their licence or insurance expires and temporarily **suspend listings** if these credentials are not renewed [5] [6]. An administrator interface shall allow compliance officers to review uploaded documents (licence, insurance certificates) and mark profiles as "verified."
- **Geographic Restrictions:** The platform must enforce state-based practice restrictions. For instance, **conveyancers are not permitted to operate in Queensland or ACT** under current law [7]. The system will warn users if a customer from a certain state attempts to engage a conveyancer not licensed for that jurisdiction, and it will prevent booking in non-allowed combinations. Conveyancer profiles should clearly list the jurisdictions where they are licensed, and searches/filters will use this info (e.g. a VIC user only sees VIC-licensed professionals by default).

Search & Marketplace Browsing

- **Conveyancer Directory & Filtering:** Customers can browse and search the marketplace of conveyancers. The system shall provide filtering by location/jurisdiction, price range (hourly or fixed), specialties (e.g. residential, commercial, off-the-plan), languages, ratings, and other attributes. Only **verified** conveyancers (completed KYC and licence check) appear in search results.
- **Profile View:** The platform displays detailed conveyancer profiles as described above. It should show license status (e.g. "Licensed in VIC, valid till 12/2025 "), insurance status, number of completed jobs on the platform, average rating, and reviews. Any verified badges or state-specific indicators (like "Licensed Conveyancer (Victoria)") are shown. For transparency, the profile also lists standard terms or sample contract, if provided.
- **SEO & Static Pages:** The frontend site should support search engine optimization for public pages (e.g. landing page, list of conveyancers in a city) via server-side rendering. Common informational pages (How it works, Fees, Terms of Service, Privacy Policy, Help/FAQ) need to be accessible without login. These content pages should be easily editable by admins (a content management requirement).

Communication & Document Exchange

- **Secure Messaging:** Once a customer has identified a conveyancer, they can initiate a **secure chat** through the platform to discuss their needs and scope out the job. The system shall provide real-time messaging (e.g. via WebSocket) allowing text chat and **file attachments**. All messages and attachments must be timestamped and stored in a conversation log for audit and dispute resolution purposes [8] [9]. The chat should only be accessible to the two parties (and moderators if needed), enforcing access control.
- **Document Upload & Virus Scanning:** Users (both customers and conveyancers) can upload documents within the chat or job workspace (e.g. property titles, contracts, IDs, certificates). The platform will utilize pre-signed URLs for direct-to-object-storage uploads from the client, and **virus scan every file** upon upload using ClamAV before making it available for download [10] [11]. Infected files are quarantined to protect users. Documents are stored in an encrypted S3-compatible storage

(using MinIO locally, and AWS S3 or equivalent in production) with versioning enabled [12] [13]. This ensures that if a file is updated (e.g. revised contract), previous versions are retained.

- **Electronic Signing:** The platform shall integrate with an **electronic signature service** (such as DocuSign or Adobe Sign) so that conveyancing documents can be signed digitally through the platform interface [14]. Most legal documents in property transactions (including contracts of sale, Section 32 statements in VIC, etc.) can be validly signed electronically under the Victorian Electronic Transactions Act and similar legislation, as long as the parties consent and required witness procedures are followed [15]. The system will allow a conveyancer to send a document for e-signature to the client (and any witnesses), track the signing status, and store the signed document. An **audit trail** of the signing process (timestamps, signers' identities, IP addresses) must be recorded for legal evidentiary purposes [16].

Job Quotes & Milestones

- **Job Creation:** When a customer is ready to engage a conveyancer, the conveyancer can formalize the engagement by creating a **Job** (contract) in the system. A Job ties a specific customer to a specific conveyancer for a defined scope of work. It includes details such as property address, property type (e.g. residential or commercial), and any special conditions. The Job's status workflow includes states like *Proposed*, *Active/Ongoing*, *Completed*, *Cancelled*, etc.
- **Scope & Milestone Definition:** The conveyancer should break the job into **milestones** (e.g. "Contract Review", "Property Searches", "Settlement") each with a description, due date, and fee amount. This milestone-based approach is required to facilitate partial payments and reduce risk [17] [18]. The platform shall allow the conveyancer to propose a quote comprising one or more milestones, which the customer can review. The customer may accept as-is or request changes via the chat. Once both parties agree, the quote is accepted, converting the milestones into an active schedule attached to the Job. Milestones have statuses such as *Proposed*, *Accepted*, *Completed*, *Released (paid)*, or *Disputed*.
- **Notifications:** The system should send notifications (in-app and email/SMS) for key events: when a quote is proposed or updated, when a milestone is due or a deliverable is uploaded, and when a milestone release is requested. This keeps both parties informed of job progress.

Payments & Escrow Management

- **Escrow Setup:** To book a job, the customer is required to pay the fee for at least the first milestone (or optionally the full job amount) upfront into an **escrow account**. The platform itself **must not directly hold client funds** to comply with Australian regulations on conveyancer trust accounts [19]. Instead, the system integrates with a **regulated Payment Service Provider (PSP)** that offers marketplace escrow functionality. When the customer authorizes a payment, the PSP will capture the funds and hold them in escrow (in a segregated client account) on behalf of the client. The PSP's API will notify the platform (via webhook) of successful funding. Each escrow payment record in the system links to the PSP's transaction reference.
- **Trust Account Payouts:** Upon milestone completion and approval, the platform instructs the PSP to release funds. **Released funds must go directly into the conveyancer's designated trust account** (an authorized deposit-taking institution account they've registered) rather than any platform-owned account [20]. The PSP should support payouts to the bank details on file for the conveyancer. This approach ensures compliance with laws requiring conveyancers to deposit client monies into trust accounts under their control, with the necessary audit trail reported to regulators [19]. The platform

will not disburse funds until the client explicitly approves the milestone or an automatic release condition is met (see Disputes). If a milestone is disputed, funds remain on hold until resolution.

- **Payment Gateway Integration:** The Payments service will integrate with the PSP's API for operations: authorizing charges, holding/capturing funds, refunds (if needed), and confirming disbursements. It must handle partial captures for each milestone and full/partial refunds if a job is cancelled. Webhooks from the PSP (e.g. payment succeeded, payout completed, chargeback notifications) are handled asynchronously by updating the relevant records and triggering notifications via the Jobs service [21] [22].
- **Fees & Invoicing:** The platform operator can charge a service fee (either to the client, the provider, or split). The system should calculate any marketplace fees and GST (Australia's 10% Goods and Services Tax) as applicable. The PSP's split-payment feature or a secondary transaction can route the fee portion to the platform's account automatically. The system must generate **GST-compliant invoices/receipts** for each payment, itemizing the conveyancer's fee and any platform fees, with tax components clearly indicated [23]. Both clients and conveyancers should be able to download payment receipts and see their transaction history for accounting purposes.

Work Execution & Deliverables

- **Task Tracking:** As the conveyancer works on the case, they can mark milestones as completed in the system and optionally attach deliverables (e.g. "Title search results uploaded" as a file attachment to the milestone). The customer is notified and can review the deliverable. For each milestone deliverable, the customer can **approve and release payment** or raise an issue.
- **Document Management:** All documents exchanged or produced during the job (contracts, IDs, property certificates, etc.) should be stored under that Job's record in a secure document repository. Each document entry tracks metadata such as uploader, upload time, file type (e.g. "Contract of Sale", "Identity Document", "Settlement Statement"), and an access URL. Access to these documents is restricted to the job's participants (and admins). The system should log every document upload and download action as part of the audit trail [16].
- **Version Control:** The document storage should allow updated versions (for example, if a contract draft goes through revisions). Users should be able to see the latest version, while previous versions remain accessible if needed for reference. This ensures transparency in any changes made to critical documents.

Completion, Reviews & Ratings

- **Job Completion:** After the final milestone (e.g. settlement) is done, the job is marked **Completed**. If there are any remaining escrow funds (for example, if all milestones were funded upfront), the system will trigger a final release of those funds to the conveyancer's account, after any waiting period or checks. The platform should implement an automatic release of escrow funds if the client does not take action within a predefined period after a completion request – this protects conveyancers from non-responsive clients (e.g., release funds automatically say 7 days after conveyancer marks milestone complete, unless a dispute is filed).
- **Review Prompt:** Upon completion, the system prompts the customer to leave a **rating and review** for the conveyancer, and optionally for the conveyancer to leave feedback on the client. The reviewing mechanism must enforce that only users who actually completed a job can review (to comply with authenticity requirements) [24]. Each completed job generally allows one review per party.

- **Ratings & Reviews:** The platform shall publish reviews on the conveyancer's profile. It must ensure **reviews are genuine** and not manipulated – the Australian Competition & Consumer Commission (ACCC) prohibits fake or misleading reviews [24]. Mechanisms include one-review-per-job, verification that the reviewer was indeed a client, and possibly a delay to allow dispute resolution before posting. A profanity filter or moderation step should be in place to prevent defamatory or inappropriate content [24]. Conveyancers should have a right-of-reply to publicly respond to any review, fostering fairness. If the platform offers any incentives for leaving reviews (e.g. a discount on next purchase), it must apply equally and be transparently disclosed, per ACCC guidance [25]. Reviews flagged as fraudulent or violating content guidelines can be removed by admin moderation.

Dispute Resolution & Support

- **Dispute Initiation:** If a client is unsatisfied with a milestone deliverable or any aspect of the service, they can flag a **dispute** before the escrow for that milestone is released. The system should allow the user to submit a dispute reason and any supporting evidence (additional documents or messages) associated with the job. Once a dispute is raised, the involved milestone's funds remain on hold.
- **Dispute Workflow:** The platform must provide an interface for managed resolution. This includes notifying the conveyancer of the dispute and allowing them to respond with their perspective or additional evidence. Each dispute could have a status (Open, Under Review, Resolved, Escalated). The platform's support/admin team should have tools to review all communication and files, then make a resolution decision (e.g. refund the client, release funds to conveyancer, or propose a compromise). The system should support **neutral adjudication** by an admin or an external arbitrator if needed [26]. All actions and decisions in a dispute are logged for accountability.
- **Service Policies:** The Terms of Service, cancellation policy, and escrow rules should be clearly accessible to users (e.g. via help pages and during checkout) to set expectations on how disputes and cancellations are handled [27]. The platform may implement an **automatic deadline** for disputes – for example, if no dispute is raised within X days of a milestone completion request, the escrow is automatically released [28].
- **Customer Support:** Users should have access to support channels (help center documentation, contact form or chat for support). The system may include an admin dashboard where support agents can impersonate user views, reset 2FA, or manually verify documents as needed to resolve user issues.

Administrative Functions

- **User & Content Management:** Administrators must have a secure back-office portal to manage the platform. They can search and view user profiles, verify or deactivate conveyancer accounts, and oversee KYC results. Admins should be able to moderate messages or documents if there are reports of abuse, and remove or edit content that violates terms (e.g. offensive chat messages or fraudulent profiles).
- **Compliance Monitoring:** The admin portal should highlight any compliance alerts – e.g. conveyancer insurance expired or about to expire, KYC verification failed for a user, a conveyancer's licence nearing expiry or found invalid upon periodic re-check. For each conveyancer, admins can log licence verification checks (date, who verified, results) and upload evidence if needed. The system should assist in **re-verifying licences and insurance annually** for each active conveyancer [6] (possibly through automated checks or reminders to upload new documents).
- **Audit Logs:** Every administrative action (such as an admin editing a profile, resolving a dispute, or accessing a user's documents) must be recorded in an **audit log** with timestamp, admin identity, and

details of the change. These logs should be stored in tamper-evident form (append-only storage) to support audits and forensic needs [29].

- **Analytics & Reports:** The platform should provide internal reports for business and compliance use. Examples: volume of transactions, escrow funds currently held, released, refunds issued, number of new users, etc. It should also generate **trust account statements** or **unclaimed monies reports** as required by state regulations [30] [31] (e.g. if client funds remain unclaimed or in escrow beyond certain periods, as per NSW/State laws). Financial reports can assist with mandatory audits (conveyancer trust accounts in NSW must be audited annually and reported to Fair Trading [19]). The system should facilitate exporting the needed data for such audits.

Technical Architecture Overview

The platform is built as a **microservice-based architecture**, using modern web technologies and containerization. The provided Docker Compose configuration defines the following components: a Next.js frontend served via NGINX, several backend microservices (Gateway, Identity, Jobs, Payments), and supporting infrastructure services (PostgreSQL, Redis, MinIO object storage, ClamAV antivirus, and an observability stack with OpenTelemetry, Prometheus, Loki, Grafana). This modular design ensures each service is focused on a set of responsibilities, and the system can be scaled or modified service-by-service. High-level design choices include:

- **Frontend Tech Stack: React** with **Next.js** (TypeScript) is used for the web frontend, enabling server-side rendering for SEO, efficient routing, and a dynamic single-page application feel [32]. The UI uses a modern component library (e.g. Radix UI or similar) and Tailwind CSS for responsive styling. Next.js runs behind an **NGINX web server** container that serves static assets and proxies API calls. NGINX can also handle SSL termination and act as a reverse proxy to backend services (or to the API gateway), consolidating external exposure to a single domain. Real-time features (chat, notifications) are enabled via WebSockets (e.g. Next.js API routes or a separate socket server). Authentication on the frontend is handled with HTTP-only cookies for JWT refresh tokens to maintain security against XSS.

- **Backend Tech Stack:** The backend is composed of multiple microservices, with a possible implementation in modern C++ (using gRPC and REST frameworks) as suggested by the design document [33]. Each microservice runs in its own container and exposes gRPC interfaces for internal communication, while a unified external REST API is exposed through an **API Gateway** layer. The rationale for gRPC internally is to achieve high performance, type-safe contracts between services, and low latency communication. Externally, REST/JSON APIs ensure compatibility with web and mobile clients. The services use a shared data layer (PostgreSQL, etc.) and share common libraries for logging, auth, etc., but are otherwise independent. Key microservices include:

- **API Gateway:** Acts as the single entry point for frontend and third-party clients to call backend functionality. The gateway routes incoming HTTP requests to the appropriate internal service (often via REST-to-gRPC calls). It handles cross-cutting concerns such as authentication (verifying JWTs on requests), rate limiting, and aggregation of responses if needed. By centralizing external API endpoints here, the microservices can operate behind the scenes and only the gateway is exposed to the public network. This also simplifies CORS and endpoint management. The gateway itself may be built using a lightweight web framework or even an NGINX config with routing rules for static

endpoints. It will define routes like `/api/auth/*` to Identity service, `/api/jobs/*` to Jobs service, etc. For development, the Next.js app can proxy `/api` calls to this gateway.

- **Identity Service:** Responsible for all user identity and profile data. This service manages user accounts (registration, login, password or OAuth if applicable), session tokens, and profile details for both customers and conveyancers [34] [35]. It integrates with the external KYC provider to verify user identities – e.g. the service will call out to an ID verification API and handle webhook callbacks for KYC results [36]. It also handles **conveyancer license and insurance verification** logic: storing the details provided, marking them verified (once checked by an admin or through an API), and enforcing any rules (e.g. cannot accept bookings if licence is expired or not verified). The Identity service issues authentication tokens (JWTs) for other services to validate. It also handles user roles/permissions (ensuring only admins can perform certain actions, etc.). Additionally, it maintains **ratings and reviews** data associated with profiles [34], and may implement functionality for user notifications (e.g. email verification, password reset). Interfaces: this service exposes REST endpoints for user registration, login, profile update, and internal gRPC APIs for other services to query user info or validate tokens.

- **Jobs & Messaging Service:** Handles the core marketplace interactions: job postings (contracts between customers and conveyancers), milestone tracking, chat messaging, and document management [37]. This service maintains the **Job** records and their associated Milestones, as well as the chat messages and attachments for each job. It provides endpoints to create a job, propose or update milestones, send and fetch messages, upload documents (which it links to object storage), and change job status. For real-time chat, this service might incorporate a WebSocket server or push notifications; it could use a publish/subscribe mechanism or in-memory websockets (possibly with a Redis-backed channel for scaling). Documents uploaded are scanned (via a ClamAV service integration) and then stored in MinIO; this service would generate pre-signed URLs for direct client uploads [11]. It uses **PostgreSQL** for persistent data (jobs, messages, etc.) and **Redis** as a message broker or cache for transient data (e.g. queued notifications, chat message fan-out) [38]. It also likely interacts with the Payments service to update job status when payments are made or released. Interfaces: REST endpoints for creating/updating jobs, posting messages, etc.; WebSocket interface for chat; gRPC or event bus for internal notifications (e.g. broadcast “milestone X released” to other services).

- **Payments & Escrow Service:** Manages all payment transactions and escrow logic in the platform [39]. It integrates with the chosen external payment gateway/escrow provider via API. Key tasks include: initiating payment authorization when a customer checks out a milestone or job (and receiving a confirmation that funds were held), listening for provider webhooks (e.g. payment succeeded, payout complete), and triggering release of funds when a milestone is approved [21]. It keeps records of all transactions, including amounts held in escrow, amounts released, refund transactions, and references to the external PSP’s identifiers for reconciliation [40]. This service also generates the invoices/receipts and handles any logic around platform fees (e.g. deducting a commission). It may interact with Identity or Jobs services via gRPC to update user balances or job statuses (for example, notify the Jobs service that “milestone 2 for Job 123 has been paid and is now in escrow”). It is also responsible for storing the **trust account details** (bank account info) of each conveyancer, which are needed for payouts – these might be encrypted and stored securely, or ideally the PSP stores them and the service just has a token reference. Interfaces: external webhooks

endpoint (for PSP -> our service notifications), REST endpoints (or through gateway) for clients to initiate a payment or refund, and internal gRPC calls to notify other services of payment events.

(Note: In a production system, the gateway and these microservices can be written in any suitable language or framework. The design suggestion was C++ for performance with frameworks like gRPC and Pistache/Crow for REST [33], but the requirement is not language-specific as long as the service contracts are met.)

- **Infrastructure Services:** Several infrastructural components are included in the architecture to support the above application logic:
- **PostgreSQL Database:** The primary relational database for persistent data storage [41]. All core entities – users, profiles, jobs, milestones, messages, payments, reviews – are stored here in a structured form. The requirement is to use a reliable ACID-compliant database to ensure transactional integrity (especially important for payments and escrow status updates). PostgreSQL is chosen for its robustness and rich feature set (JSON support, etc.). The schema should be designed to enforce referential integrity (e.g. milestones linked to jobs, jobs to users). We expect moderate sizing initially (hundreds to thousands of users and jobs), but the DB should be able to scale vertically and be optimized with indexes for key queries (such as searching profiles by location or retrieving a chat history quickly).
- **Redis Cache/Queue:** A Redis service is used for caching and transient data [41]. It can store session data or JWT blacklists if needed, cache frequent read queries (to lighten DB load on expensive operations), and implement rate-limiting counters. Redis also serves as a lightweight message broker or task queue – for example, to queue background jobs like sending notification emails or processing audit log batch inserts. The Jobs service might use Redis Pub/Sub or streams to broadcast new chat messages to other server instances. Requirements for Redis are to operate as an in-memory datastore for speed, with persistence (RDB or AOF) as needed for recovery of critical ephemeral state.
- **Object Storage (MinIO/S3):** All uploaded files (documents, images, etc.) are stored in an object storage service rather than in the database, due to their size and to allow streaming. For development and self-hosting, **MinIO** (an S3-compatible storage server) is included via Docker Compose. In production, an equivalent cloud storage (like AWS S3, Azure Blob Storage, or Google Cloud Storage) will be used. Requirements: files must be stored securely (encrypted at rest), with access controlled by signed URLs (so that only authorized users can download). There should be a retention policy on certain types of documents depending on legal requirements (see Data Retention section). MinIO provides an S3 API which our application uses via SDK to upload/download. The platform should implement a **virus scanning** step on each file upload – e.g., the file is uploaded to a temporary bucket/key, scanned by **ClamAV** before being moved to a permanent location or marked as safe [11]. If ClamAV (running as a service container) finds malware, the file should be rejected and not accessible.
- **ClamAV Antivirus:** The architecture includes a **ClamAV** service container which updates virus definitions and scans files for malware. The requirement is to integrate this scanning either at upload time (synchronously) or via a background job that quarantines files until they are scanned. Given sensitive personal and legal documents will be exchanged, this protects users and the platform from known malicious files. ClamAV is invoked either via a direct command-line call through a shared volume or network socket by the document service.

- **Message Broker (Kafka/RabbitMQ):** For certain asynchronous workflows and event-driven communication, the system should use a message broker. The architecture document suggests Kafka or RabbitMQ for events such as “payment processed” or “milestone released” [41]. This is not

explicitly listed in the Docker Compose snippet but is a recommended part of the tech stack. The requirement is to reliably queue and broadcast events that multiple services might need to react to. For example, when the Payments service confirms a payout, it could emit an event that the Jobs service listens to in order to mark the job as paid. Similarly, KYC verification results might come via events. A lightweight option is to use Redis pub/sub if Kafka is too heavy for MVP, but a scalable solution would include Kafka for high throughput and persistence of event logs.

- **Observability & Ops Stack:** To meet reliability and performance goals, the platform includes tools for monitoring, logging, and tracing:

- **OpenTelemetry (Tracing):** All services are instrumented with OpenTelemetry to trace requests across service boundaries. For example, a single user request that goes through the Gateway to the Identity service and then to the Jobs service will carry a trace ID, allowing developers to see a unified trace in debugging tools. The requirement is to capture distributed traces to diagnose latency or failures in complex workflows [42]. The OTEL collector can be run as part of the docker-compose to gather traces; these can be viewed via Grafana Tempo or exported to a SaaS APM as needed.
- **Prometheus (Metrics):** A Prometheus service is deployed to scrape metrics from the microservices and infrastructure. Each service will expose an endpoint (e.g. `/metrics`) with Prometheus client metrics – such as request counts, error counts, database query timings, memory usage, etc. Custom business metrics will also be recorded (e.g. number of new jobs created per day, active chats, escrow release latency) [43]. Prometheus is configured to alert on certain conditions (via Alertmanager), for example high error rate, high latency, or low available DB connections. The system should define SLOs (service level objectives) and set up corresponding alerts (see **Performance & Availability Targets**).
- **Loki (Logging) and Grafana (Dashboard):** For centralized logging, the platform can use Grafana Loki (a log aggregation system) or ELK (Elasticsearch) stack [41]. In our architecture, Loki is included for simplicity. Each service outputs structured logs (JSON with fields like timestamp, level, trace ID, message) to STDOUT, which Docker captures and Loki can pull. This allows searching across logs of all services in one place. The logs are crucial for auditing and debugging; they must be retained for a certain period (e.g. 7+ days in hot storage, archived later). Grafana ties everything together by providing dashboards for metrics and an interface to query logs and traces. The requirement is that ops engineers can quickly visualize system health (CPU, memory of containers, request rates, error rates, etc.) and correlate events via trace IDs. Grafana will also be used to create business dashboards (e.g. daily transactions, user signups).

- **Correlation and Auditing:** Each request should be assigned a correlation ID (e.g. the trace ID) that is included in logs across services [44]. This makes it easier to follow a single transaction through multiple services in the logs. Additionally, sensitive actions (like admin actions, financial transactions) should be logged in append-only audit logs as mentioned, possibly stored in a secure location (WORM storage).

- **Deployment & Containerization:** All components (frontend, backend services, DB, etc.) are containerized via Docker. The Docker Compose configuration allows running the entire suite on a development machine or single host for testing. For production deployment, these containers would be orchestrated using a platform like **Kubernetes** or a container service (ECS, Docker Swarm). The system should be cloud-agnostic to an extent, relying on standard images for Postgres, Redis, etc., and environment variables for configuration (database URLs, API keys, etc.). Each microservice will

have its own codebase and Docker image; continuous integration should produce images tagged with versions or commit SHAs, and continuous deployment will roll out updates service-by-service. This architecture ensures that the platform can scale horizontally by adding more instances of stateless services behind load balancers, and critical stateful components (DB, etc.) can be scaled vertically or clustered as needed.

Security, Privacy, and Compliance

Authentication & Access Control

Security is paramount due to the sensitive nature of transactions and personal data. The platform will implement robust authentication and authorization measures: - **Authentication:** Users log in via secure methods – either traditional email/password (with strong password policy and bcrypt hashing) or OAuth2/ social login, or **passwordless** magic links for convenience. Sessions are managed with short-lived JWT access tokens and refresh tokens (stored HttpOnly) [45] . All API calls require a valid token; the API Gateway will validate JWTs and enforce scopes/roles. Refresh tokens can be rotated and revoked on logout. 2FA as noted is available and required for conveyancers and admins. Device fingerprinting or anomaly detection should be in place to flag unusual login attempts (e.g. new device or location).
- **Role-Based Authorization:** The system uses role-based access control (RBAC) and possibly attribute-based access for fine-grained permissions. Roles include Customer, Conveyancer, and Admin (plus potential sub-roles like Support Agent). Sensitive endpoints (e.g. verify another user's document, or release someone else's funds) are restricted to admins. Conveyancers can only view data for jobs they are involved in, and customers only their own transactions. Until a booking is confirmed, personally identifiable information of users should be limited – e.g. a conveyancer shouldn't see a client's full name/contact until the client books them, to protect privacy and prevent off-platform solicitation [46] . Admin actions require elevated privileges and possibly additional login steps.
- **Session Security:** JWTs will be short-lived (e.g. 15 minutes access token) to reduce risk, with refresh tokens (http-only, secure cookies) for maintaining sessions [45] . The system will implement token revocation for compromised accounts (e.g. via a token blacklist in Redis or using JWT with short TTL so revocation delays are minimal). Additionally, rate limiting on login and sign-up APIs will be enforced to prevent brute force or spam (e.g. using Redis counters) [47] .

Data Security & Privacy

- **Encryption in Transit and At Rest:** All communications must use TLS (SSL) with modern protocols (TLS 1.2+). This includes the frontend to backend calls (HTTPS) and internal service-to-service calls (which can be within a secure network, but it's recommended to use TLS for gRPC as well). For data at rest, the database and object storage must employ encryption. Database-level encryption includes using encrypted volumes and optionally field-level encryption for particularly sensitive fields (e.g. passwords are hashed, but also things like identity document numbers or bank account numbers can be encrypted using application-level keys so that even DB dumps are protected) [35] . Backups of the database should also be encrypted.
- **Privacy by Design:** The platform will follow Australian **Privacy Principles (APPs)** to ensure personal information is handled lawfully. Personal data is collected only as needed for the service (data minimisation) and used for the stated purposes. Users must consent to data collection, and a privacy policy will outline usage. **Data retention** is limited to what's necessary: Australian privacy law mandates not retaining personal info longer than required [48] . In practice, given other regulations

require 7-year retention for certain records, the system will enforce deletion or anonymization of personal data once it's no longer legally or operationally needed (see Data Retention section). The right to erasure (right to be forgotten) is honored for users who request account deletion, insofar as it doesn't conflict with legal record-keeping obligations [6].

- **Data Residency:** There is no absolute law forcing local storage of all data in Australia; however, if any user data is stored in overseas data centers, the platform remains responsible under the Privacy Act to ensure equivalent protection (APP 8) [49]. The system will likely use an Australian region (e.g. AWS Sydney) for primary data storage to inspire user trust and simplify compliance. Highly sensitive data (financial info, identity documents) will be stored in infrastructure within Australia whenever feasible [49]. If using any third-party services, we will ensure they are either located in Australia or accredited under schemes like the Australian Government's IRAP or similar, or we have proper consent from users for offshore data processing.
- **Confidentiality & Access Controls:** All user data access in the application is governed by the principle of least privilege. Service-to-service interactions use authentication (e.g. internal API keys or mutual TLS) so that one service cannot accidentally read another's data without permission. Within the database, separate schemas or tables for different services can limit blast radius. Administrative access to production data (e.g. via database or dashboards) should be tightly controlled – staff accounts require 2FA and access logs must be kept. Logs and monitoring dashboards should avoid exposing personal data openly; for instance, logs might redact or hash sensitive identifiers when not necessary for debugging.
- **Audit Trails:** Security also involves accountability. The system shall maintain detailed audit logs for critical events: user logins, important configuration changes, payments, document views/downloads, etc. These logs should be **immutable** or append-only (to prevent tampering) [29]. Audit logs will capture "who, what, when, and where (IP/device)" for actions and be stored in secure storage separate from the main database (possibly in the log aggregation system with proper retention). This is important for both security forensics and compliance audits (e.g. demonstrating to regulators that proper processes are followed).

Regulatory Compliance

Australia's legal landscape for conveyancing and online marketplaces imposes several requirements that the system must adhere to:

- **Conveyancer Licensing:** It is a legal requirement that any service provider offering conveyancing is licensed in their state. The platform must **verify the licence of each conveyancer** before they can offer services [3]. This involves checking the official registers (for example, NSW Fair Trading's public register under the Conveyancers Licensing Act 2003, or Consumer Affairs Victoria's register) [3]. The system will store each provider's licence number, issuing authority, and expiry. It should automatically flag or prevent a conveyancer from taking new jobs if their licence is expired or if a regulator removes their licence. Conveyancers are not allowed to operate in certain jurisdictions (e.g. QLD, ACT) [7], so the software must enforce those restrictions as described in Functional Requirements. Regular checks (e.g. via an admin routine or an API if available) should be done to re-validate licences annually or more frequently. All licence verification actions should be logged.
- **Professional Indemnity Insurance:** Conveyancers in most states must hold active professional indemnity insurance (in VIC, at least $2M coverage) and must cease practice if insurance lapses [4]. The platform will collect insurance information and proof from providers and will not allow a profile to remain active if the insurance is not verified or is expired. This means when a conveyancer's insurance expiry date passes, their account might automatically be suspended from new work until

updated (with notifications sent in advance). Storing the insurance details and expiry in the system enables these checks [4].

- **Trust Account & Escrow Compliance:** By law, conveyancers must use trust accounts for client funds and adhere to strict handling and reporting rules. The platform itself **should never directly hold client money** to avoid being seen as an unlicensed intermediary [19]. Instead, the requirement is to use a **marketplace payment provider with escrow capability** so that funds can be held in a compliant way (the conveyancer's own trust account or a fully regulated escrow on behalf of them). For example, when a client funds a milestone, the money goes into a segregated account with an authorized ADI (bank) and is only released as directed by the platform's instructions in accordance with client approval [20]. The system must support capturing the necessary details for each transaction (who paid, when, which trust account it's destined for). Also, every deposit to a trust account may require notifications/forms to regulators (in NSW, each trust deposit needs an official notice to Fair Trading) [50] – while the platform can delegate this to the conveyancer or the PSP, it should maintain records that facilitate compliance (like keeping track of deposit references). The escrow integration should also support refunds and cancellations, since those must return funds to the client if a job doesn't proceed.
- **KYC/AML Obligations:** Conveyancing services fall under **anti-money-laundering (AML)** scrutiny. The platform operator likely qualifies as providing a service that may be considered a "reporting entity" if it handles certain financial transactions, thus it must implement KYC/AML programs. Concretely, as mentioned, all users must undergo identity verification. The platform will **record verification data for at least 7 years** to comply with record-keeping rules [1]. According to AUSTRAC and AML legislation, we must retain identification and transaction records for seven years and report any suspicious matters or large cash transactions to AUSTRAC (though large transactions in this context are unlikely) [51]. The system should be able to flag unusual behavior (e.g. repeated large payments, mismatches in identity) for manual review. Additionally, part of KYC for conveyancers includes verifying their professional credentials (licence, insurance) as those contribute to validating the legitimacy of providers [52]. The system will store evidence of identity checks and allow retrieval of this evidence for audit by regulators if required.
- **Electronic Transactions Act Compliance:** Since the platform will facilitate electronic signing of documents, it must comply with relevant electronic transactions laws (which are largely similar federally and by state). In Victoria, for example, electronic signatures are broadly valid, including for contracts and even deeds (with conditions for witnessing remotely) [15]. The platform should obtain consent from users to transact electronically (likely via the terms of service agreement). The e-signature integration must produce a compliant signing log and ensure signers are properly authenticated (the KYC process helps here). If any documents require in-person witnessing that can't be done online, the platform should flag that (though Vic now allows audiovisual-link witnessing for deeds [15]). Essentially, the system must ensure that the **digital signature process yields legally enforceable documents**.

- **Consumer Protection (ACCC) and Fair Use:** As noted, the platform must enforce honest practices. This includes **genuine reviews only** (no fake reviews, and no cherry-picking positive ones) [24]. It also means transparency in fees and no misleading representations of conveyancers' qualifications or pricing. The ACCC's guidance on online marketplaces would apply – all terms and fees should be clearly disclosed to users upfront. The platform's design should avoid any "dark patterns" or deceptive UI that could mislead users. If the platform offers promotions or incentives (like "$50 off your first milestone"), it must honor them fairly and disclose conditions. The system should also allow users to easily access dispute resolution and not unduly lock them in to bad outcomes (e.g. releasing funds without consent). Compliance with Australian Consumer Law is a broad requirement

that the system should support by design (clear information, confirm actions for payments, allow cooling-off where legally required, etc.).

- **Records Retention:** Multiple regulations (from AUSTRAC, to ARNECC for e-conveyancing, to general business law) require retaining certain records for set periods. The platform is required to **retain transaction records, identity verification data, and other evidence for at least 7 years** [53] [30]. This influences data storage design – the system cannot automatically purge old transaction data before 7 years, and must ensure backups or archives exist to fulfill this. At the same time, privacy considerations mean that after this period, data should be disposed of. A retention schedule will be implemented to archive and eventually delete data when allowable (see Data Retention section). The system should also allow exporting of data if needed (for example, if a user or regulator requests all records related to a particular job or user, we should be able to provide it).

By embedding these compliance requirements into the system design from the start, the platform will reduce legal risk and build trust with users [54] [55]. Where appropriate, the system will provide prompts or warnings to users to help them comply as well (e.g. warning a conveyancer to only use their trust account for payments, or reminding a client to read the contract cooling-off rights). Compliance features may also include maintaining up-to-date reference data (like a list of banned practitioners or sanctions lists for AML, which could be cross-checked via the KYC provider). Finally, the platform's legal terms and privacy policy will reflect all these obligations clearly to users.

Data Storage and Retention Policies

Data Storage Strategy: All data is stored in secure, redundant systems appropriate to its type. The **primary data store** is the PostgreSQL database, which holds structured records (users, jobs, milestones, payments, messages metadata, etc.). Binary large files (documents uploaded, images) are stored in the object storage (MinIO/S3) and referenced by URL in the database. Some transient or derived data is kept in caches (Redis) or logs.

- **Personal Data Fields:** Within PostgreSQL, personal identifiable information (PII) such as user full names, contact details, identity document numbers, and bank account numbers, are stored in dedicated fields. Wherever feasible, the platform uses encryption for these sensitive fields at rest (either via the database's encryption or at the application level with keys). This ensures that even if the raw database were accessed, critical PII remains protected [56]. The keys for any field-level encryption are stored securely (e.g. in a secrets manager or environment variable not in code).
- **File Storage:** Documents uploaded (which may contain personal data or legal info) are saved in a private bucket on MinIO/S3. They are **not publicly accessible**; the application provides signed URLs for download that expire after a short time. The storage buckets implement server-side encryption. We also enable versioning on the bucket to prevent accidental loss – if a file is deleted or overwritten, an older version can be retrieved by admins if needed, until purged by retention rules.
- **Backups:** Regular backups of the PostgreSQL database and critical storage are mandated. For example, a nightly full backup of the database and weekly snapshots of the storage should be taken. Backups will be encrypted and stored off-site (in a different AWS region or a secure backup service) to ensure disaster recovery capability [57]. A retention policy for backups might keep daily backups for e.g. 30 days, monthly backups for a year, and yearly for 7 years, aligning with the legal retention needs. Restore tests should be performed periodically to verify that backups are viable (this is part of operational requirements).

- **Logging data:** Logs collected in Loki (or any logging system) will include audit logs and application logs. Because logs can contain snippets of personal data (e.g. user IDs, maybe names in some context), they should be treated carefully. The log system is configured to **redact sensitive fields** (like passwords or credit card numbers, which should never appear in logs anyway by design) [29]. Logs will be retained in the hot system for a shorter period (say 30 days for easy search in Grafana), and then archived (e.g. exported to object storage or a long-term log archive) for the remainder of the retention period required. Access to logs is limited to authorized personnel.

Retention Policy: The platform implements a data retention schedule that balances legal obligations with data minimization principles:

- **User and KYC Data:** Identity verification records (copies of IDs, verification reports, licence and insurance documents, etc.) are retained for at least **seven (7) years** from the date of collection or last active use [53] [58]. This aligns with ARNECC Model Participation Rules and AUSTRAC requirements for record-keeping. After 7 years of inactivity (e.g. user has not logged in or no transactions in 7 years), the system may purge or anonymize these records, unless a regulatory investigation or legal hold requires longer retention. Account profiles that are deleted by users will be deactivated, but key verification info is stored until the retention period expires, then securely destroyed.
- **Transaction Records:** All job and payment records are kept for a minimum of 7 years as well [30]. This includes the details of the job contract, messages (since they could contain agreements or evidence), and payment history. Even if a user deletes their account, the transaction records are archived for the required period (with personal identifiers pseudonymized if necessary, but an ability to re-identify if lawfully required). Financial records (invoices, escrow logs) might be needed for tax and audit purposes and are retained per standard accounting practice (often 5-7 years in Australia). The system will have an **archival process** – e.g. moving old completed jobs to an "archived" state in the DB or to a separate archive DB after some years, to reduce load on the main system while still preserving the data.
- **Document Retention:** Uploaded documents (contracts, IDs, etc.) should similarly be kept for 7 years **from the completion of the transaction** or the date of upload [58]. After that, the documents can be deleted from storage. (For example, a conveyancing file might legally need to be kept for 7 years, after which it can be destroyed. The system can automatically purge documents older than 7 years, perhaps running a cleanup job monthly). Prior to deletion, it might optionally notify the users (though not legally required). The deletion process must be thorough (removing all versions of the file and any backups beyond retention).
- **Audit Logs & Monitoring Data:** Security audit logs should be kept long enough to investigate incidents. Many organizations keep detailed audit logs for 1 year online and a few years offline. Given our regulatory environment, audit logs for critical actions might also be kept for 7 years to match user records. Less critical system logs can be rotated more frequently (e.g. debug logs for only 30 days). The retention settings will be configured accordingly in Loki/ELK and backup storage.
- **Right to Erasure:** If a user requests deletion of their data (and if they have no ongoing jobs), the platform will comply by deleting personal data from active systems – but certain information must be retained if it falls under legal retention (the system would inform the user that, e.g., "We have deleted your profile, but we must retain transaction records for 7 years as required by law"). In practice, personal identifiers in those retained records could be anonymized (for example, replace user name with a unique code, while keeping a separate secure mapping if re-identification is ever needed under legal authorization). The system will document these procedures in the privacy policy

and ensure the engineering implementation supports extracting or removing user data on request (data portability and deletion tools).

- **Data Disposal:** When the retention period is met, the system should permanently destroy the data. For database records, this means deletion or overwriting; for files, secure deletion. Any encryption keys associated with the data should also be disposed of if that renders the data unrecoverable. Proper logging of data deletion actions (what was deleted, when) should be kept as a meta-record (perhaps a simple log that doesn't itself contain the data, just references).

By adhering to these storage and retention policies, the platform will safeguard user information, support business needs, and comply with regulations like the Privacy Act and ARNECC rules. It ensures that data isn't kept longer than necessary, reducing liability surface over time.

Performance, Scalability, and Availability

Performance Targets

The system is expected to provide a smooth, responsive experience for users even during peak usage. Key performance targets include: - **Response Time:** Most user-facing API calls (for example, loading a list of conveyancers or sending a chat message) should respond within **300ms** to **500ms** under normal load (not counting network latency to the user). Heavy operations like searching numerous profiles or generating a PDF invoice might take a bit longer but should still aim for sub-second times. Real-time interactions (typing in chat, receiving messages) should feel instantaneous – the WebSocket pathway should handle message round-trip in a few hundred milliseconds at most on average.
- **Concurrent Users and Throughput:** The MVP should handle on the order of **hundreds of concurrent active users** (simultaneous sessions) initially, with design headroom to scale to thousands as the platform grows. In terms of throughput, the system might see dozens of new jobs created per day in early stages, scaling to hundreds per day with multi-state operation. The infrastructure should handle, say, **100 chat messages per second** or **50 payments per hour** with ease, scaling further via horizontal scaling. The microservices and database will be load-tested to support these rates. - **Database Performance:** Queries should be optimized so that common pages (dashboard, profile search) can load quickly. Using proper indices and caching, we aim for most queries to complete in under 50ms. Any long-running operations (like generating aggregate reports, or nightly compliance checks) should be done asynchronously or during off-peak hours to avoid blocking the user transactions.

To achieve these performance goals, the system employs techniques such as query optimization, in-memory caching (Redis for frequent lookups like authentication sessions or reference data), and efficient protocols (gRPC binary communication internally). Web content is optimized via Next.js features (SSR caching, code splitting for fast initial loads). We will conduct **load testing** (using tools like JMeter or k6) to verify that the system meets target QPS (queries per second) and to identify bottlenecks. Performance profiling will be part of the development cycle, especially for the C++ services, ensuring that serialization, database access, etc., are efficient.

Scalability

The architecture is designed for horizontal scalability. Each stateless service (Gateway, Identity, Jobs, Payments, frontend server) can be replicated behind load balancers to increase throughput. In a cloud deployment, we would run multiple instances of each microservice in a Kubernetes cluster or auto-scaling

groups. The gateway can distribute incoming REST calls to multiple backend instances. WebSocket scalability for chat can be achieved through sticky sessions or a message broker to propagate messages to all relevant server nodes.

- **Microservice Scaling:** For example, if the chat message volume increases, we can scale out the **Jobs & Messaging** service to more instances to handle additional WebSocket connections and message throughput. If API calls to identity (login/registration) spike, the **Identity** service instances can be increased. Because services communicate via gRPC/HTTP, adding instances should be transparent to others (service discovery or a fixed routing configuration will be used).
- **Database Scaling:** The PostgreSQL database can be vertically scaled (move to a larger instance with more CPU/RAM and faster I/O) as the data grows. Additionally, we can introduce read replicas for load distribution (especially for heavy read scenarios like analytics or search queries) – the application can direct read-only queries to replicas. For write scaling beyond a single instance's capacity, partitioning or sharding could be considered, but that likely won't be needed until the user base is very large. In later stages, if needed, a migration to a distributed SQL or a sharded setup could be planned.
- **Storage and Other Services:** The MinIO/S3 storage inherently scales by design (S3 is virtually unlimited; MinIO can be clustered if needed). Redis can be scaled by vertical growth or using clustering/sharding if high volumes of cache data or pub/sub throughput are required. The Kafka/RabbitMQ, if used, should be set up in a clustered mode for high availability and to handle high event rates.
- **Stateless vs Stateful:** We aim to keep services stateless where possible (i.e. not relying on in-memory sessions that can't be reconstructed). This way, any instance can handle a request and if an instance dies, it doesn't hold unique state. Sessions are stored in Redis or cookies so that users don't lose their session if one frontend instance goes down. File uploads go to shared storage, not local disk. This stateless design eases horizontal scaling and rolling deployments.

The architecture also considers multi-tenancy for the future (if the platform hosted multiple marketplace instances or something similar), but initially it's a single tenant (Australia-wide marketplace). If needed, separate instances for different states or regions could be deployed and later integrated, but we prefer scaling the single platform as it expands to new states with configuration differences handled in software (not separate deployments).

Availability & Reliability

High availability is crucial for a marketplace handling financial transactions. Downtime or data loss can severely impact users and trust. The platform targets an availability of at least **99.9% uptime** (which equates to < ~9 hours downtime per year), with a stretch goal of **99.95%** as the system matures. To achieve this:

- **Redundancy:** In production, every critical component will have redundant instances. For example, we'll run the database in a primary-hot standby setup (e.g. AWS RDS Multi-AZ for PostgreSQL) so that if one instance fails, the standby can take over with minimal interruption. Similarly, at least two instances of each microservice will run so that one failing does not bring down the service – the load balancer will route traffic to the healthy instance. The MinIO storage can be replaced by AWS S3 which is highly durable (11 nines durability) and available across multiple AZs. Redis can be run with a replica and using Redis Sentinel or a managed service for failover. The goal is to eliminate single points of failure.

- **Failover and Recovery:** The system should support graceful degradation. For instance, if the Payments service is down for a few minutes, users should still be able to browse and chat, and any payment actions queued to retry when it's up. We will implement health checks and circuit breakers – e.g. if a microservice is unreachable, the gateway can return a friendly error or fallback (instead of hanging). The infrastructure will have automated **restart policies** for containers and possibly self-healing via orchestration (Kubernetes will reschedule failed pods). Disaster Recovery (DR) procedures will be in place: in case of a total outage in one region, data backups and infrastructure-as-code can spin up the environment in another region within a target Recovery Time Objective (RTO) of a few hours. The database backups and storage snapshots ensure Recovery Point Objective (RPO) is low (possibly < 1 hour of data loss in worst case).
- **Monitoring & Alerts:** As described, the monitoring system will actively track uptime and error rates. We will define SLAs such as: "95th percentile API response < 1s" and "Error rate < 0.1%". If these thresholds are breached or if any service becomes unresponsive, the system (via Prometheus Alertmanager or an external service) will alert engineers. On-call procedures will be established to respond 24/7 to critical outages. Additionally, automated scaling can be configured for some components (e.g. if CPU on all app servers is >80% for 5 minutes, scale out another instance) to handle unexpected load bursts and maintain availability.
- **Transactional Integrity:** Reliability also means no lost or inconsistent data, even in failure scenarios. We utilize database transactions to ensure, for example, that when a payment is marked released, the milestone status update and invoice generation all succeed or all fail together. In cross-service workflows, a saga pattern or compensation mechanism will handle partial failures (for instance, if a payment succeeds but updating the job status fails, the system can recover by reconciling with the PSP's records and fixing the status when the service is back up). We prefer at-least-once processing for critical events, even if it means sometimes processing a duplicate (with idempotent handlers) rather than dropping an event.
- **Testing & Quality Assurance:** To maintain high availability, changes to the system will be thoroughly tested (unit, integration, and end-to-end tests in staging environments) before deployment. Continuous integration will run test suites on every code change, and we will employ canary releases or blue-green deployment techniques in production so that new updates do not take down the system (deploy new version alongside old, verify health, then switch traffic). This reduces the risk of outages due to deployment issues.

With these measures, the platform aims to provide a reliable service where users can confidently engage in transactions at any time. In summary, downtime will be minimized by redundancy and quick failover, and the system will be built to handle increasing load by scaling out smoothly.

Development and Deployment Environment

Development Environment

For local development and testing, developers will use Docker Compose to run the full stack on their machines. The repository includes a `docker-compose.yml` that defines all required services (frontend, backend services, database, etc.) with their interdependencies. The recommended developer workstation should have at least **16 GB of RAM** and a modern multi-core CPU (e.g. 4+ cores) to comfortably run the

containers. This is because running PostgreSQL, Redis, MinIO, ClamAV, and multiple microservices can be resource intensive.

- **Local Setup:** Developers can bring up the entire environment with a single command (`docker-compose up`) which will start the NGINX + Next.js frontend, the API gateway, the Identity service, Jobs service, Payments service, PostgreSQL, Redis, MinIO, ClamAV, OpenTelemetry collector, Prometheus, Loki, and Grafana. Environment variable configurations (like database URL, API keys for third-party integrations in sandbox mode) are provided via an `.env` file or Docker Compose variables. The local environment uses development settings – e.g. reduced load, dummy third-party integrations if needed (for instance, a mock KYC service and a dummy payment provider that simulates escrow without actual money movement, for safe local testing).
- **Developer Workflow:** Code changes to the microservices or frontend can be hot-reloaded or require rebuilding the container depending on language. For example, with Next.js, developers can run it in dev mode on host for fast refresh, while backend C++ services might be recompiled and the Docker images rebuilt. To streamline this, the project can include scripts or use volumes to reflect code changes inside containers (for interpreted languages or Node, this is straightforward; for C++, one might compile on host and just restart the service container). The development environment should mirror production architecture as closely as possible, but with conveniences like seeding test data (e.g. a few dummy users) and verbose logging.
- **Testing:** Unit tests can be run outside of containers for speed, but integration tests might run with the whole docker-compose environment up. The team might use additional tooling like Testcontainers or a specialized testing compose file to bring up services for automated test runs (for CI). Developers should also be able to run an individual service with the others pointing to it (e.g. run Identity service directly in an IDE for debugging, while the rest run in Docker, connecting via network). The environment should facilitate that by exposing necessary ports and using host networking or Docker networks accordingly.
- **Source Control and CI:** The source code will be maintained in a git repository, with separate folders for each microservice and the frontend. A continuous integration (CI) pipeline (e.g. GitHub Actions, GitLab CI, Jenkins) will build and test the code on each commit. The CI will also build Docker images and could push them to a registry (like ECR, Docker Hub) with a tag corresponding to the commit or version. This ensures consistency between dev and prod builds.

In summary, the local dev environment aims to be one-command reproducible, using Docker Compose to avoid “it works on my machine” issues. Sufficient hardware is needed to run it smoothly, and developers are guided to use environment config that parallels production (such as using the same environment variables names, etc.), but connecting to sandbox/test accounts for external APIs.

Deployment (Production Environment)

For production (and staging) deployment, a robust cloud environment is recommended. The system will likely be hosted on a cloud provider such as AWS, Azure, or GCP to meet scalability and reliability requirements. Key aspects of the deployment environment include:

- **Container Orchestration:** We will use a container orchestration platform, most likely **Kubernetes**, to deploy and manage the microservices in production. Kubernetes allows easy scaling, self-healing, and rolling updates. Each microservice will run in a deployment with a desired number of replicas. Kubernetes Services or an API gateway ingress will route traffic. (Alternatively, AWS ECS or Fargate

could be used, but Kubernetes offers more flexibility for the observability stack integration). The observability components (Prometheus, Grafana, etc.) can also run in the cluster or be provided by managed services.

- **Compute Resources:** An initial production cluster might consist of 3 application nodes (VMs or cloud instances), for example each with 4 vCPUs and 16 GB RAM, to distribute the load and provide redundancy across availability zones. These nodes would host the containers for the gateway and microservices. The database is typically not containerized in production but rather using a managed service (e.g. Amazon RDS for PostgreSQL) for better reliability. Similarly, Redis could be an AWS ElastiCache instance (with primary/replica). MinIO would be replaced by AWS S3 service, eliminating the need to manage storage servers.
- **Networking & Security:** The deployment will be within a virtual private cloud (VPC). Services that don't need public exposure (database, internal microservice ports) will be kept in a private subnet. Only the NGINX/web gateway (or a load balancer in front of it) is exposed on the internet. We will obtain an TLS certificate for the domain (e.g. via ACM or Let's Encrypt) to serve HTTPS. Security Groups/Network ACLs will restrict access (for example, only the app servers can talk to the database host on its port; only the load balancer can talk to app ports, etc.). Kubernetes network policies can add an extra layer of isolation between services. Secrets (like API keys for third-party services, database passwords, JWT signing keys) will be stored in a secure secrets manager or as Kubernetes secrets, not in plain config files.
- **Scaling & Load Balancing:** In front of the web tier, a cloud load balancer (like ALB or Nginx ingress controller with multiple replicas) will distribute incoming traffic. It will perform health checks on the backend services. Auto-scaling rules can be set: e.g. if CPU > 70% on average, add another replica. The DB instance can be scaled vertically (e.g. from a 2 vCPU to 4 vCPU machine) as load grows, and read replicas can be added to offload read-heavy operations. We plan for horizontal scaling for stateless services and careful scaling for stateful ones.
- **Monitoring & Logging in Cloud:** We will deploy Prometheus and Grafana in the cluster for full control, or use managed solutions (e.g. Amazon CloudWatch for metrics, but since the stack is set up for Prom/Grafana, we likely continue with that). Grafana can run as a deployment with persistent storage for its config (or use Grafana Cloud SaaS). Logs from applications can be sent to a centralized log service; if using Loki, we might run Loki in cluster with persistent volume, or use a cloud log service (like CloudWatch Logs or Elastic Cloud). Traces from OpenTelemetry can be sent to a backend like Jaeger, Tempo, or a SaaS (like New Relic, Datadog) depending on preference. The key requirement is that any solution used must cover metrics, logs, and traces with alerting, as in dev.
- **Software Dependencies:** In production, all services run as Docker images built from the code. We need to ensure the base images are secure (use official images for Node, C++ runtime, etc., and keep them updated for patches). The environment should also account for region-specific configurations: for instance, using AWS Sydney region endpoints for services, time zone settings to AEST for logs, etc. Content delivery (like serving the Next.js static files) can be aided by a CDN (CloudFront or similar) to speed up global access, though initially users are mostly in Australia so a regional focus is fine.
- **Resource Allocation:** For initial sizing, each microservice container might be allocated something like 0.5 vCPU and 512MB-1GB RAM request (scaling up if needed). The Next.js server might need 1 vCPU/1GB. Prometheus and Grafana will use a few GB of RAM for metrics. ClamAV, when running scans, is CPU intensive but runs sporadically. The hardware planning ensures that peak usage (like multiple file scans and many users) doesn't exhaust resources. Overprovisioning a bit at start is wise to handle growth spurts.
- **Continuous Deployment:** We will set up a CI/CD pipeline so that when changes pass tests and are merged, new container images are deployed to a staging environment automatically. After further

testing, a production deployment can be triggered (possibly manually approved for now). Deployments are done with zero downtime strategies (rolling update in Kubernetes). Configuration for different environments (dev/staging/prod) is handled via environment variables and Helm charts or similar, keeping sensitive values out of code. The deployment scripts/infrastructure (Infrastructure as Code using Terraform or CloudFormation or Kubernetes manifests) will be maintained to allow reproducible environment setup and to track changes.

By using modern cloud infrastructure, the platform will achieve reliability and ease of maintenance. New developers can spin up their own copy of the environment via Docker Compose, and production issues can be debugged by comparing with staging or using the robust logging/tracing in place. The approach ensures that from local dev to production, the differences are minimized (e.g., using S3 in prod vs MinIO in dev is abstracted by the S3 API; using a dummy payment API in dev vs real in prod is just a config switch). This consistency reduces surprises and accelerates development.

Integration Points

The platform must integrate with several external systems and services to provide its full functionality and comply with regulations. Key integration points include:

- **Payment Service Provider (Escrow Payments):** As described, the platform partners with a PSP that can hold funds in escrow and perform split disbursements. Likely candidates include Stripe (with Connect accounts for conveyancers and PaymentIntents for escrow-like flows), Assembly Payments (an Australian escrow/payments API, now known as Zai), or a similar fintech service. The integration involves using the PSP's API to create payment charges, authorize holds, capture funds to the conveyancer's account, and issue refunds. Webhooks from the PSP must be handled to get asynchronous updates (e.g. a payout succeeded or failed). This integration requires secure storage of API keys and compliance with PCI-DSS (though using a third-party means we mostly redirect customers to a secure checkout or tokenize card details so the platform's servers don't see full card numbers). The PSP integration will also handle compliance aspects like anti-fraud checks and maybe AUSTRAC reporting of large transfers. **Regulatory note:** By routing payments to trust accounts via the PSP, we align with legal escrow requirements [20], but we still must ensure record-keeping on our side (which the integration facilitates by providing transaction logs).

- **Identity Verification (KYC) Service:** To automate KYC for user onboarding, the platform will integrate with a third-party **Digital Identity verification** provider. Options in Australia include Australia Post **Digital iD**, OCR Labs, Stripe Identity, Equifax ID, or GreenID (SecureKey) among others. These services typically provide an API where we send user-provided info (and perhaps images of documents or live selfies), and they return a verification result. Some, like Digital iD, allow the user to complete the verification through a redirect flow for security. The chosen service should be compliant with Australian Document Verification Service (DVS) checks and have an AML compliance standard. The integration will involve sending data like name, address, DOB, document IDs to the provider and receiving verification status and possibly an ID reference. We will also set up **webhook callbacks** from the provider if the verification is not instant (some take minutes to manual check). The system marks the user as KYC-verified once a success callback is received [36]. If the KYC fails or flags issues (e.g. document is fake or person is politically exposed), the system should handle that (e.g. require manual review, inform the user). This integration ensures we meet the KYC

obligations for AML [1] without building a solution from scratch. The KYC provider must be reputable and preferably has an SDK or integration guide for a smooth user experience.

- **License Registry APIs / Checks:** There is no single nationwide API for conveyancer licenses; each state has its own register (some public web portals, not all with open APIs). Integration here might be partly automated and partly manual. For example, NSW Fair Trading might have a data service or at least a nightly data dump of licensed conveyancers. If available, the platform will connect to such an API to validate licence numbers in real-time when a conveyancer signs up or periodically (the PDF references using public registers [3]). If APIs are not available, the system may instead assist admins by providing quick links or even web scraping scripts to look up a licence. At minimum, our system stores the licence info and an admin confirmation checkbox that they verified it on a certain date. In future, if government open data improves, we can automate daily checks (e.g. a script that checks all active conveyancers against the register to catch any newly suspended licences). Integration here is thus semi-automated. It's a crucial point for compliance that we integrate this verification step into the onboarding workflow (even if via manual process).

- **Electronic Signature Provider:** To enable e-signing of documents, we integrate with a provider like **DocuSign, Adobe Sign,** or a local Australian alternative (e.g. HelloSign or Nitro Sign). DocuSign is a common choice with a robust API. The integration flow: when users need a document signed, our system sends a request to the e-sign API, including the document file (which might be already on our storage, so a download link is provided) and metadata about signers (names, emails) and signature fields. The e-sign provider then handles sending emails to parties or providing an embedded signing UI. Once all parties sign, the provider calls a callback in our system or we poll for status. The signed PDF is then retrieved via the API and stored in our system. The integration must also handle edge cases like a signer declining or a signature expiring (not completed in X days). This external service ensures signatures are legally compliant with necessary audit trail. The platform will include in its UI status updates like "Pending signature from client" and allow users to click a link to sign (which goes through the provider's secure interface). We must store the evidence file (signature certificate) provided. By using a reputable e-sign service, we ensure compliance with the Electronic Transactions Act (since these providers often have templates that meet legal standards) [14].

- **Communication Services:** To support notifications, the platform will integrate with email and possibly SMS services. For example, integrate with an **email API service** (like SendGrid, Amazon SES, or Mailgun) to send verification emails, milestone alerts, etc. Similarly, an **SMS gateway** (like Twilio or an Australian SMS service) can be used for sending OTP codes for 2FA or important reminders (e.g. "Your milestone is ready for approval"). These integrations are relatively straightforward: the platform generates the content and calls the provider's API to send. It must handle delivery failures (e.g. invalid email) by logging or notifying the user to update contact info. Ensuring emails are not flagged as spam (proper SPF/DKIM records) is part of this integration's responsibility. These are not unique to this platform but are necessary for full functionality.

- **Analytics and Tracking:** Though not a core part of requirements, typically a platform might integrate with analytics or marketing tools (e.g. Google Analytics, or a customer engagement platform) to track usage and conversion. If needed, we ensure any such integration complies with privacy (get user consent for cookies, etc.). For a system requirements perspective, this is optional.

Each integration must be secured (API keys stored safely, use HTTPS), and failure of an integration should not break the whole system. For example, if the KYC service is down, conveyancer sign-up might be queued or given a “pending verification” status rather than completely failing. If the PSP is down, we should disable new payments temporarily with a user-friendly message. We will implement graceful error handling around all external API calls. Additionally, we should monitor the usage limits and costs of these integrations (to ensure we don’t exceed API quotas or incur unexpected fees).

Regulatory Citations in Integration: The design of these integration points is heavily influenced by compliance: e.g. using a regulated payment service and not handling money directly is in response to trust account laws [20]; performing KYC with an official provider is to meet AUSTRAC’s requirements [1]; using a certified e-signature service is to comply with e-sign legislation [14]; verifying licences via government sources is mandated by state regulations [3]. Each of these external systems has been chosen or designed into the architecture to satisfy those legal requirements while delivering a seamless user experience.

Conclusion: The above system requirements detail a comprehensive plan for building a secure, compliant, and efficient conveyancing marketplace platform in Australia. By combining robust functional features (milestone payments, chat, document handling) with strong non-functional foundations (scalable microservices, security by design, thorough observability), the platform will be well-equipped to launch in Victoria and gradually expand. Adhering closely to regulatory obligations – from verifying licences and identities to handling client funds correctly – is not only a legal necessity but a competitive advantage in building user trust [59]. With this architecture and requirements in place, the development team can proceed confidently to implementation, knowing that the critical system qualities and compliance features have been specified from the outset.

---

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59

Marketplace platform architecture.pdf

file://file_0000000098186206b08e7fdcae5bc8c9