



Technical Architecture Document

Complete system design reference for the AI-powered proof review platform

Version 1.0 | February 2026 | Internal

Table of Contents

Table of Contents	2
Executive Summary	4
Key Technology Choices	4
System Architecture Overview	5
Technology Stack	5
Multi-Agent Analysis Pipeline	7
Specialist Agents	7
Lead Agent RAG Decision Logic	7
Revision-Aware Analysis	7
WebSocket Analysis Flow	9
Message Protocol	9
Flow Lifecycle	9
Database Schema	11
Domain Overview	11
Key Design Decisions	11
Frontend Architecture	13
Component Hierarchy	13
Views	13
State Management	13
Authentication & RBAC	15
Authentication Flow	15
Role Hierarchy	16
Backend Enforcement	16
Knowledge Base Pipeline	18
Knowledge Base Types	18
Processing Pipeline	18

Version Management	18
Deployment Architecture	20
Infrastructure Components	20
Deployment Process	20
API Reference Summary	22
REST Endpoints	22
Knowledge Base Endpoints	23
WebSocket Endpoint	23
Appendix A: Environment Variables	24
Backend (backend/.env)	24
Frontend (frontend/.env.local)	24
Appendix B: Technology Stack	25
Backend Dependencies	25
Frontend Dependencies	25

Executive Summary

ModComms is an AI-powered proof review tool built for Barclays marketing operations. It automates the compliance, brand, tone-of-voice, and channel-suitability review of marketing assets (proofs) that would traditionally require manual review by multiple specialist teams.

The platform employs a **multi-agent AI architecture** where four specialist agents analyse each proof in parallel, with a lead agent synthesising their findings into a single RAG (Red/Amber/Green) status decision. This enables near-instant feedback on uploaded proofs, dramatically reducing review cycle times.

Key Technology Choices

- **Frontend:** React 18 SPA with TypeScript, Vite build tool, Tailwind CSS
- **Backend:** Python FastAPI with async/await, WebSocket real-time communication
- **AI Engine:** Google Gemini 2.5 Flash for multi-modal image analysis
- **Database:** PostgreSQL with SQLAlchemy async ORM and Alembic migrations
- **Authentication:** Azure AD via MSAL with 4-tier RBAC
- **Knowledge Base:** LlamaParse document parsing + Gemini distillation pipeline

System Architecture Overview

ModComms follows a **three-tier architecture** with a React single-page application frontend, a Python FastAPI backend, and PostgreSQL for persistence. The backend communicates with external services including Google Gemini for AI analysis, Azure AD for authentication, and LlamaParse for document processing.

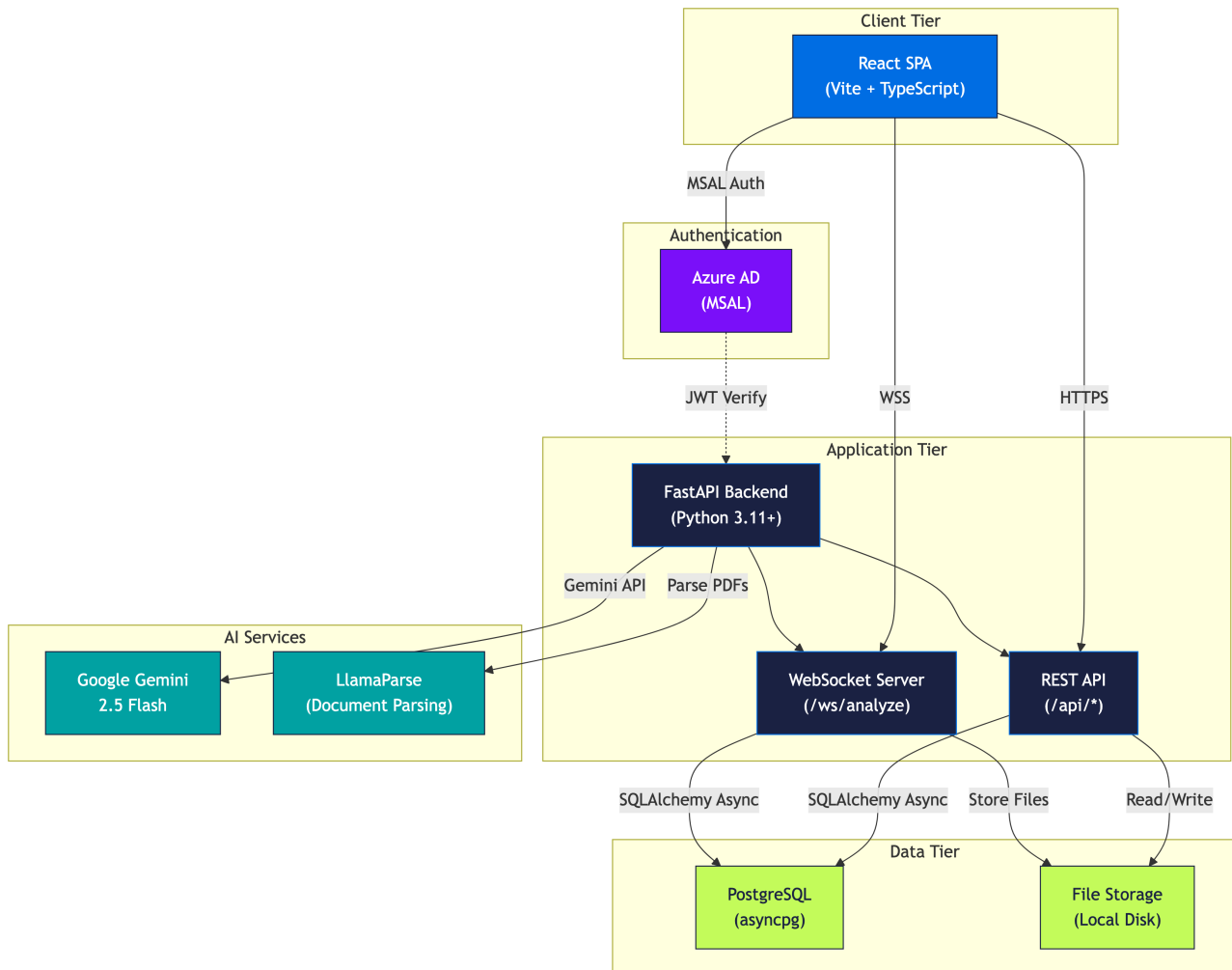


Figure 1: High-Level System Architecture

Technology Stack

Tier	Technology	Purpose
Frontend	React 18 + TypeScript	Single-page application UI
Frontend	Vite 5	Build tool and dev server
Frontend	Tailwind CSS	Utility-first styling
Frontend	@azure/msal-react	Azure AD authentication
Backend	Python 3.11+ / FastAPI	Async REST API + WebSocket server

Tier	Technology	Purpose
Backend	SQLAlchemy 2.0 (async)	ORM with asyncpg driver
Backend	Alembic	Database migration management
Backend	Uvicorn	ASGI server
AI	Google Gemini 2.5 Flash	Multi-modal proof analysis
AI	LlamaParse	PDF/DOCX document parsing
Database	PostgreSQL 15	Primary data store
Auth	Azure AD / MSAL	OAuth2 + JWT identity provider
Deployment	Docker Compose	Container orchestration
Deployment	Apache2 + mod_proxy	Reverse proxy and static serving

Multi-Agent Analysis Pipeline

The core of ModComms is a multi-agent system where four specialist agents analyse each proof in parallel. Each agent has a distinct area of expertise and access to a curated knowledge base of reference documents. After all agents complete, a Lead Agent synthesises their findings.

Specialist Agents

Agent	Focus Area	Key Checks
Legal Agent	Advertising standards compliance	Financial promotion detection, required disclaimers, FCA/ASA rules, risk language
Brand Agent	Brand identity adherence	Logo usage, colour palette, typography, design language principles (Barclays or Barclaycard)
Channel Best Practices Agent	Platform-specific guidelines	Content best practices, accessibility, readability, platform conventions
Channel Tech Specs Agent	Technical specifications	Dimensions, file size limits, format requirements, resolution, safe zones

Lead Agent RAG Decision Logic

The Lead Agent synthesises all four sub-reviews into an overall RAG status and a human-readable summary:

- **GREEN (Passed):** All agents pass with at most 1 amber-level issue per agent, and no Legal agent amber issues.
- **AMBER (Requires Manual Legal Review):** More than 1 actionable issue per agent, or any Legal agent amber-level issue.
- **RED (Failed):** Any agent returns a Red status, indicating a critical compliance or brand violation that must be resolved.

Revision-Aware Analysis

When a proof has been previously analysed (version $N > 1$), the system automatically fetches the prior version's analysis results and passes them as context to each agent. This enables agents to identify **resolved issues**, **outstanding issues**, and **new issues** relative to the previous version, providing actionable delta feedback.

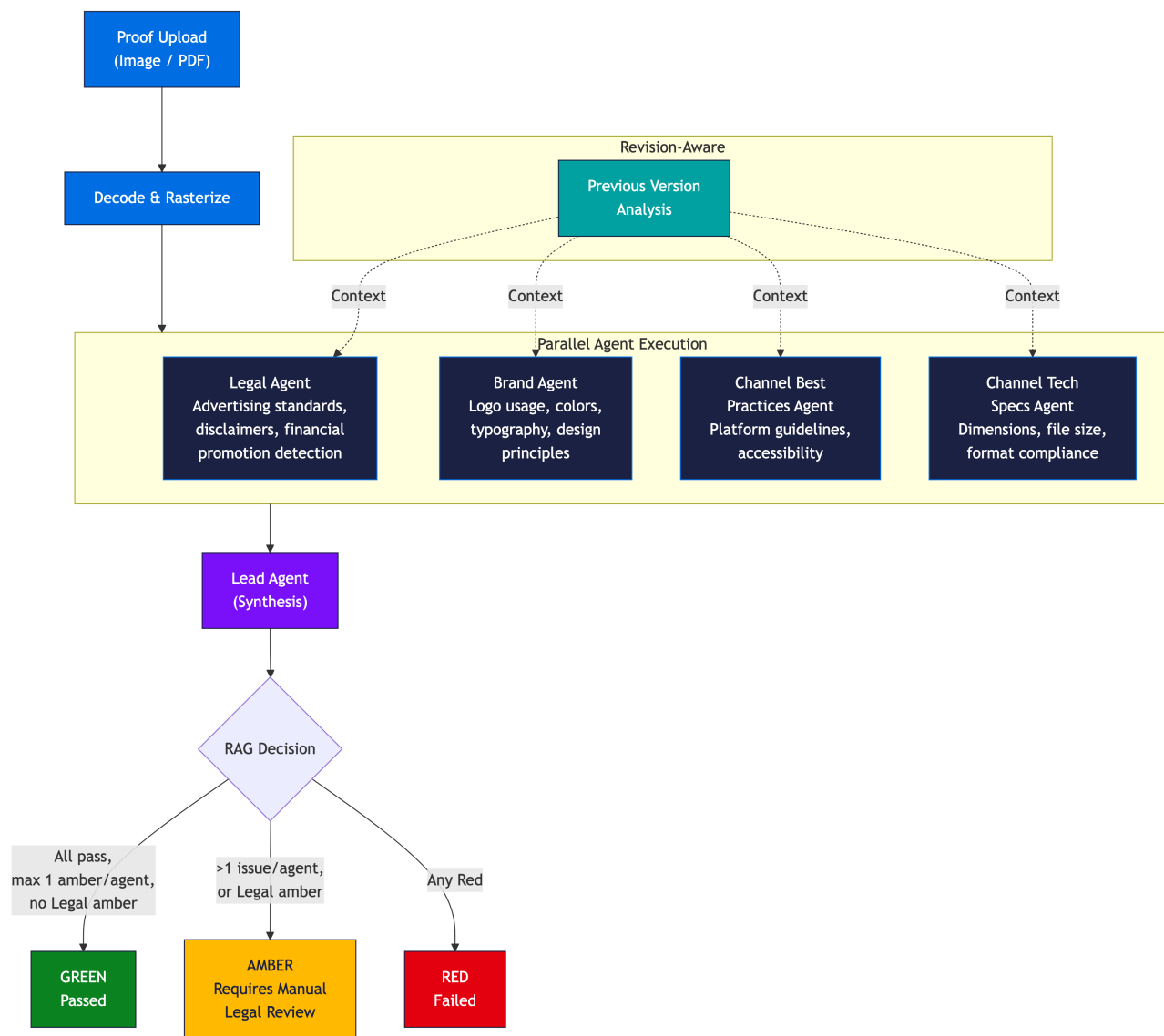


Figure 2: Multi-Agent Analysis Pipeline

WebSocket Analysis Flow

Proof analysis uses a WebSocket connection for real-time streaming of agent progress. The client sends a single `analyze` message containing the base64-encoded file and metadata, and receives a stream of updates as each agent starts and completes.

Message Protocol

Direction	Type	Payload	Description
Client → Server	<code>analyze</code>	<code>file_data</code> , <code>file_type</code> , <code>access_token</code> , <code>brand</code> , <code>campaign_id</code> , <code>proof_name</code> , <code>channel</code> , <code>sub_channel</code> , <code>proof_type</code>	Initiate analysis with file and metadata
Server → Client	<code>agent_started</code>	<code>agent_name</code>	Agent has begun processing
Server → Client	<code>agent_completed</code>	<code>agent_name</code> , <code>review</code> (<code>ragStatus</code> , <code>feedback</code> , <code>issues</code> , <code>resolvedIssues</code> , <code>outstandingIssues</code> , <code>newIssues</code>)	Agent finished with results
Server → Client	<code>complete</code>	<code>result</code> (all agent reviews + lead summary + overall status), <code>proof_id</code> , <code>version_id</code> , <code>is_identical_file</code> , <code>pdf_pages</code>	Full analysis complete and persisted
Server → Client	<code>error</code>	<code>message</code>	Error occurred during processing

Flow Lifecycle

- 1. Client establishes WebSocket connection to `/ws/analyze`
- 2. Client sends `analyze` message with JWT access token
- 3. Server verifies JWT token against Azure AD
- 4. Server checks user role (`oversight_admin` blocked from analysis)
- 5. Server decodes base64 file data; rasterizes PDF pages if applicable
- 6. Server fetches previous version analysis for revision context
- 7. Four agents run in parallel via `asyncio.gather()`
- 8. Real-time `agent_started` / `agent_completed` messages stream to client
- 9. Lead Agent synthesises overall RAG status
- 10. Results persisted to database; file stored to disk
- 11. `complete` message sent with full results and IDs

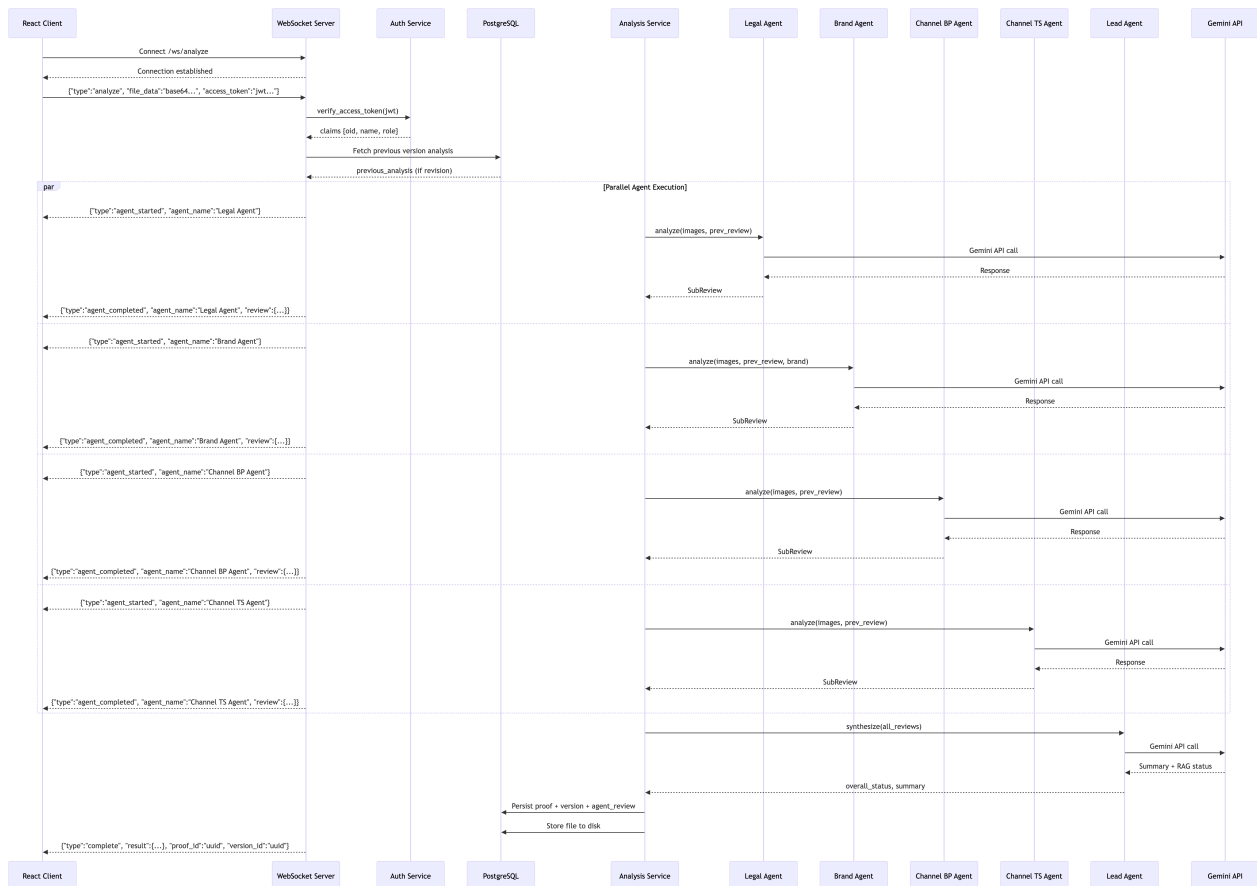


Figure 3: WebSocket Analysis Sequence

Database Schema

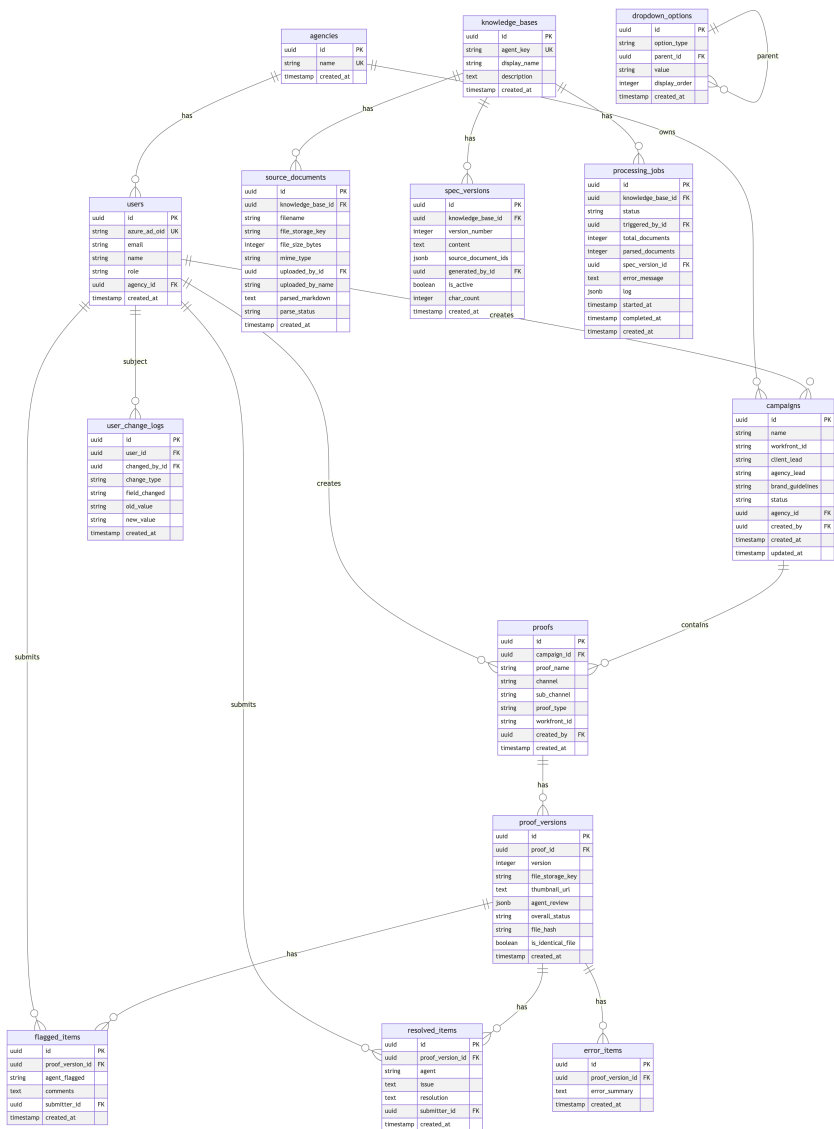
ModComms uses PostgreSQL with SQLAlchemy 2.0 async ORM. The schema comprises 15 tables organised into four logical domains. All primary keys are UUIDs. Alembic manages migrations.

Domain Overview

Domain	Tables	Purpose
Identity & Access	agencies, users, user_change_logs	User accounts, agency membership, role audit trail
Campaign & Proof	campaigns, proofs, proof_versions	Marketing campaigns, proof assets, versioned analysis results
Audit & Review	flagged_items, resolved_items, error_items	Manual flagging, issue resolution tracking, analysis error records
Knowledge Base	knowledge_bases, source_documents, spec_versions, processing_jobs	Agent reference documentation, document processing pipeline
Configuration	dropdown_options	Hierarchical channel/sub-channel/proof-type configuration

Key Design Decisions

- **JSONB for agent_review:** The `proof_versions.agent_review` column stores the complete multi-agent analysis result as a JSONB document, enabling flexible querying while keeping the schema stable as agent output evolves.
- **JSONB for processing logs:** `processing_jobs.log` stores step-by-step pipeline progress as structured JSON for debugging.
- **Self-referential hierarchy:** `dropdown_options` uses a `parent_id` FK to itself, supporting the Channel → Sub-Channel → Proof Type hierarchy in a single table.
- **File hash deduplication:** `proof_versions.file_hash` (MD5) detects when an identical file is re-uploaded, flagging it to the user.



Frontend Architecture

The frontend is a React 18 single-page application built with TypeScript and Vite. It uses Tailwind CSS for styling with a custom Barclays design system. The application is wrapped in an MSAL authentication provider and a custom UserContext for role-based rendering.

Component Hierarchy

The entry point (`index.tsx`) wraps the app in `MsalProvider`. `App.tsx` acts as an authentication gate, rendering `Login` for unauthenticated users and `AppContent` (inside `UserProvider`) for authenticated users. Views are rendered based on a `currentView` state variable.

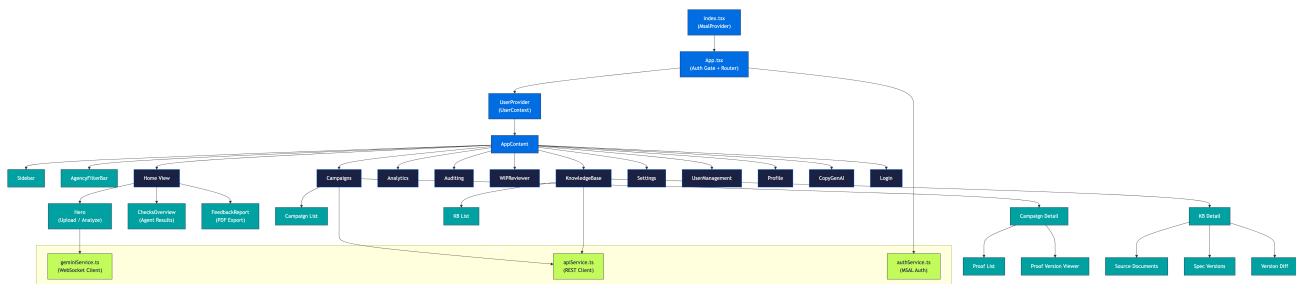


Figure 5: Frontend Component Hierarchy

Views

View	Component	Description
Home	Hero + ChecksOverview + FeedbackReport	Upload proof, view real-time analysis, export report
Campaigns	Campaigns	Campaign CRUD, proof list, version history, re-analysis
Analytics	Analytics	Aggregate statistics, RAG distributions, per-agency breakdowns
Auditing	Auditing	Flagged items, resolved items, error items with filters
WIP Reviewer	WIPReviewer	Quick analysis without persisting to a campaign
Knowledge Base	KnowledgeBase	Manage agent reference docs, trigger processing, view specs
Settings	Settings	Dropdown options (channels, sub-channels, proof types)
User Management	UserManagement	Role assignment, agency assignment, change history
Profile	Profile	Current user info and preferences
CopyGenAI	CopyGenAI	AI-assisted marketing copy generation

State Management

- **UserContext:** Provides authenticated user info, role checks (`canWrite`, `canSeeAnalytics`, etc.), and agency filtering state.
- **URL State:** Campaign and proof selections are encoded in the URL hash for deep linking and browser history support.
- **API Service:** Centralized REST client (`apiService.ts`) that auto-attaches MSAL access tokens to all requests.
- **WebSocket Service:** `geminiService.ts` manages the WebSocket lifecycle for proof analysis, including reconnection and progress callbacks.

Authentication & RBAC

ModComms uses Azure Active Directory for authentication via the MSAL (Microsoft Authentication Library) protocol. The frontend acquires tokens via MSAL.js popup flow, and the backend verifies JWT tokens using Azure AD's JWKS endpoint.

Authentication Flow

- 1. User navigates to app; MSAL checks for existing session
- 2. If not authenticated, Login component triggers `loginPopup()`
- 3. Azure AD returns ID token + access token
- 4. Frontend calls `GET /api/me` with Bearer token
- 5. Backend verifies JWT, extracts claims (oid, name, email)
- 6. Backend auto-provisions user on first login as `basic_user`
- 7. `UserContext` stores profile and computes role-based feature flags

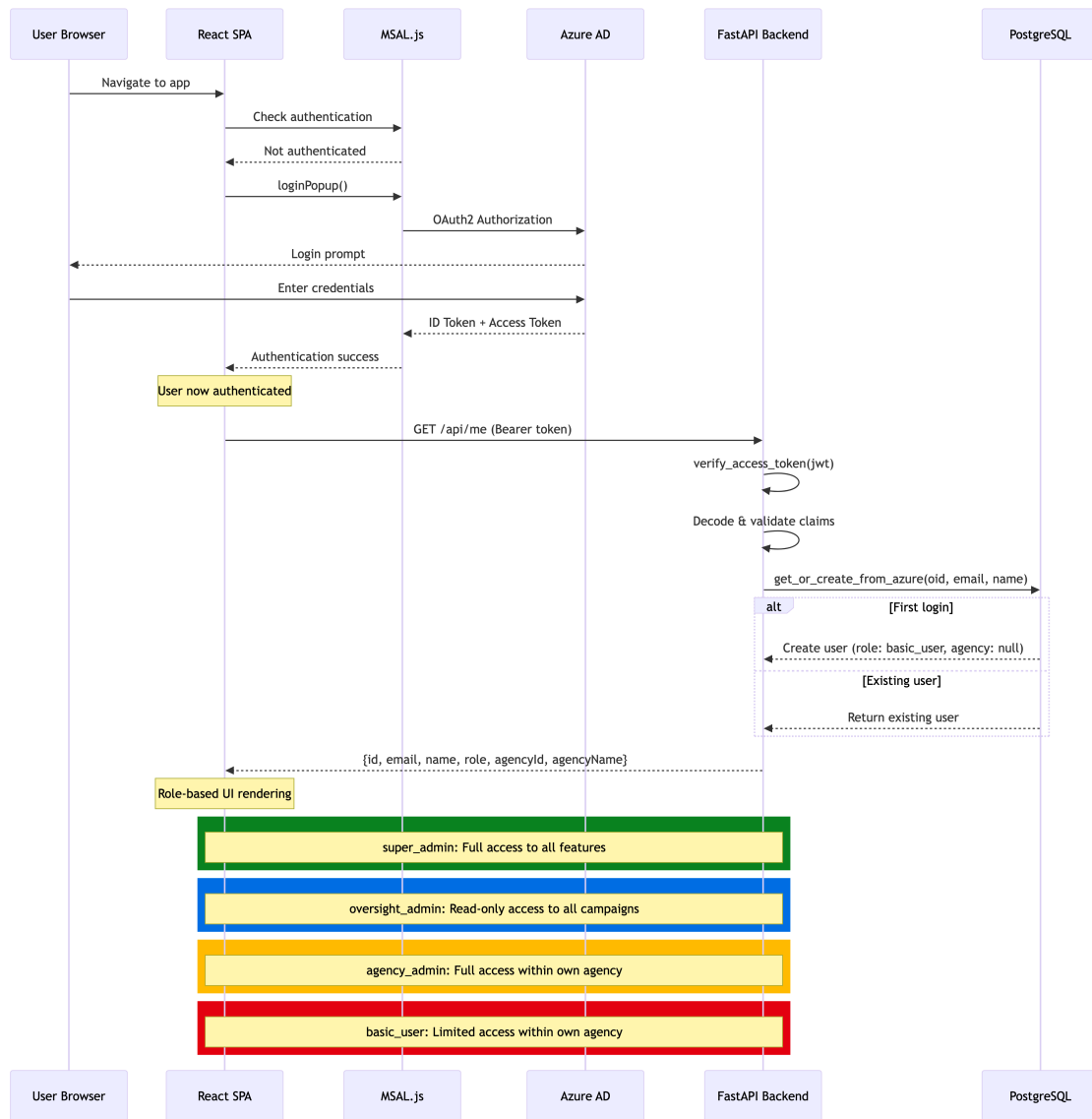


Figure 6: Authentication & RBAC Flow

Role Hierarchy

Role	Scope	Key Permissions
super_admin	Global	All features, all campaigns, user management, settings, knowledge base, analytics, auditing
oversight_admin	Global (read-only)	View all campaigns across agencies, analytics, auditing. Cannot upload, analyse, flag, or resolve.
agency_admin	Own agency	Full CRUD within own agency's campaigns, flagging, resolving
basic_user	Own agency	Upload and analyse proofs, view own agency's campaigns, flag issues

Backend Enforcement

Role-based access is enforced at the FastAPI dependency level using composable dependencies:

- `get_current_user()` — Verifies JWT and returns claims dict
- `get_current_db_user()` — Resolves claims to a User ORM object with agency
- `require_role(*roles)` — Factory that restricts endpoints to specific roles
- `require_write_access()` — Blocks oversight_admin from mutation operations

Knowledge Base Pipeline

The Knowledge Base system allows admins to upload reference documents (brand guidelines, legal standards, channel specifications) that are parsed, combined, and distilled into concise agent specifications. These specs are versioned and serve as the primary context for each agent during proof analysis.

Knowledge Base Types

Agent Key	Display Name	Description
legal	Legal Compliance	Advertising standards, FCA rules, disclaimer requirements
brand_barclays	Barclays Brand	Barclays brand identity guidelines, design language
brand_barclaycard	Barclaycard Brand	Barclaycard-specific brand guidelines
channel_best_practices	Channel Best Practices	Platform content guidelines, accessibility standards
channel_tech_specs	Channel Tech Specs	Technical dimensions, file formats, resolution specs

Processing Pipeline

- 1. **Upload:** Admin uploads source documents (PDF, DOCX, etc.) via the KB UI
- 2. **Store:** Files saved to `kb_storage/` with metadata in DB
- 3. **Parse:** LlamaParse API converts documents to clean Markdown
- 4. **Combine:** All parsed Markdown from source documents is concatenated
- 5. **Distil:** Gemini generates a concise, structured agent spec from the combined text
- 6. **Version:** New spec version created with version number, character count, and source doc links
- 7. **Activate:** New version set as active; reference docs cache invalidated
- 8. **Serve:** Agents load the active spec version at analysis time

Version Management

Spec versions are immutable once created. Admins can view the full content of any version, compare two versions with a unified diff view, and revert to a previous version by activating it. Processing jobs track the full pipeline lifecycle with status progression: `pending` → `parsing` → `distilling` → `completed` (or `failed`).

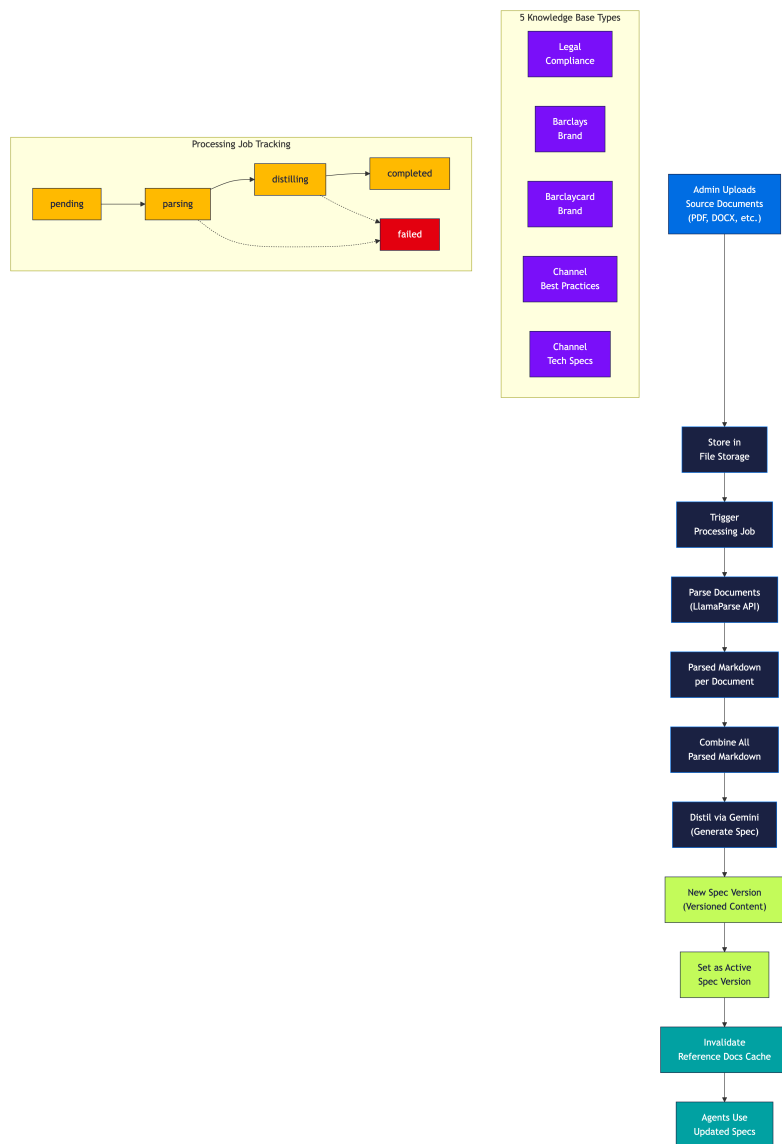


Figure 7: Knowledge Base Processing Pipeline

Deployment Architecture

ModComms is deployed using Docker Compose for the backend services and Apache as a reverse proxy serving the static frontend build. The deployment script (`deploy.sh`) automates the full build and deployment process.

Infrastructure Components

Component	Technology	Configuration
Reverse Proxy	Apache2 + mod_proxy + mod_proxy_wstunnel	HTTPS termination, static file serving, proxy to backend (port 8000)
Backend	Docker container (Python + Uvicorn)	Port 8000, auto-restart, volume mounts for uploads and KB storage
Database	Docker container (PostgreSQL 15)	Port 5432, persistent volume for data, health checks
Frontend	Static files (Apache DocumentRoot)	Vite production build served directly by Apache
File Storage	Host filesystem volumes	uploads/ for proof files, kb_storage/ for knowledge base documents

Deployment Process

- 1. **Pull:** Fetch latest code from Git repository
- 2. **Frontend build:** `npm run build` produces static assets
- 3. **Deploy frontend:** Copy build output to Apache DocumentRoot
- 4. **Backend build:** `docker compose build` rebuilds backend image
- 5. **Database migration:** `alembic upgrade head` inside container
- 6. **Restart services:** `docker compose up -d` restarts backend + DB
- 7. **Reload Apache:** `systemctl reload apache2` picks up config changes

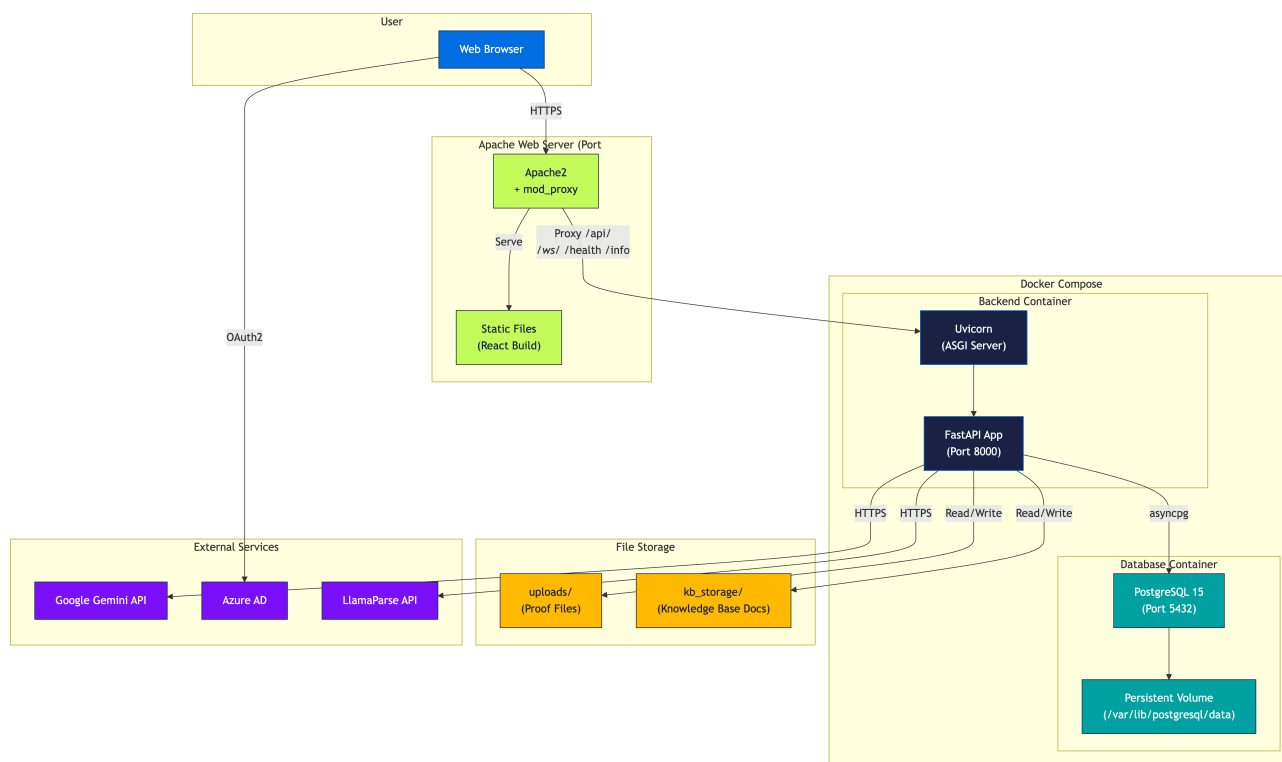


Figure 8: Deployment Architecture

API Reference Summary

REST Endpoints

Method	Endpoint	Auth	Description
GET	/api/me	Bearer	Get authenticated user profile
GET	/api/campaigns	Bearer	List campaigns (filtered by role/agency)
POST	/api/campaigns	Bearer + Write	Create a new campaign
GET	/api/campaigns/{id}	Bearer	Get campaign by ID
PUT	/api/campaigns/{id}	Bearer + Write	Update a campaign
DELETE	/api/campaigns/{id}	Bearer + Write	Delete campaign and all files
GET	/api/campaigns/{id}/proofs	Bearer	List proofs for a campaign
GET	/api/proofs/{id}	Bearer	Get proof by ID
DELETE	/api/proofs/{id}	Bearer + Write	Delete proof and files
POST	/api/proofs/{id}/versions/{v}/flag	Bearer + Write	Flag an issue
POST	/api/proofs/{id}/versions/{v}/resolve	Bearer + Write	Resolve an issue
GET	/api/audit/flagged	Bearer	List flagged items
GET	/api/audit/resolved	Bearer	List resolved items
GET	/api/audit/errors	Bearer	List error items
GET	/api/analytics	Bearer + Admin	Get analytics data
GET	/api/analytics/by-agency	Bearer + Admin	Per-agency analytics
GET	/api/users	Bearer + Admin	List all users
PUT	/api/users/{id}	Bearer + Super	Update user role/agency
GET	/api/users/{id}/change-history	Bearer + Admin	User change audit trail
GET	/api/agencies	Bearer	List all agencies
POST	/api/agencies	Bearer + Super	Create agency

Method	Endpoint	Auth	Description
GET	/api/dropdown-options	Bearer	Get channel/sub-channel/type options
POST	/api/dropdown-options/...	Bearer + Super	Manage dropdown options
GET	/api/files/{key}	Bearer	Retrieve stored file
GET	/api/files/{key}/pages	Bearer	Rasterize PDF to page images
POST	/api/support/email	Public	Send support email

Knowledge Base Endpoints

Method	Endpoint	Description
GET	/api/knowledge-base	List all knowledge bases
GET	/api/knowledge-base/{id}	Get KB detail with docs and active spec
POST	/api/knowledge-base/{id}/documents	Upload source document
DELETE	/api/knowledge-base/{id}/documents/{doc_id}	Remove source document
POST	/api/knowledge-base/{id}/process	Trigger processing pipeline
GET	/api/knowledge-base/{id}/jobs/{job_id}	Get processing job status
GET	/api/knowledge-base/{id}/versions	List spec versions
GET	/api/knowledge-base/{id}/versions/{v_id}	Get spec version content
GET	/api/knowledge-base/{id}/versions/{a}/diff/{b}	Diff two spec versions
POST	/api/knowledge-base/{id}/versions/{v_id}/activate	Activate a spec version

WebSocket Endpoint

`ws://host/ws/analyze` — Real-time proof analysis with streaming agent progress updates. See the WebSocket Analysis Flow section for full protocol details.

Appendix A: Environment Variables

Backend (backend/.env)

Variable	Required	Description
GEMINI_API_KEY	Yes	Google Gemini API key for AI analysis
DATABASE_URL	Yes	PostgreSQL connection string (asyncpg)
AZURE_TENANT_ID	Yes*	Azure AD tenant ID for JWT verification
AZURE_CLIENT_ID	Yes*	Azure AD application (client) ID
CORS_ORIGINS	Yes	Comma-separated allowed CORS origins
LLAMA_CLOUD_API_KEY	No	LlamaParse API key (enables KB pipeline)
DISABLE_AUTH	No	Set 'true' for local dev without Azure AD
REFERENCE_DOCS_PATH	No	Path to reference docs directory (default: reference_docs/)

* Required when `DISABLE_AUTH` is not true

Frontend (frontend/.env.local)

Variable	Required	Description
VITE_BACKEND_URL	Yes	Backend REST API base URL (e.g. http://localhost:8000)
VITE_BACKEND_WS_URL	Yes	Backend WebSocket URL (e.g. ws://localhost:8000/ws/analyze)
VITE_AZURE_CLIENT_ID	Yes	Azure AD app client ID for MSAL
VITE_AZURE_TENANT_ID	Yes	Azure AD tenant ID for MSAL
VITE_AZURE_REDIRECT_URI	Yes	OAuth2 redirect URI

Appendix B: Technology Stack

Backend Dependencies

Package	Version	Purpose
fastapi	0.115+	Web framework (REST + WebSocket)
uvicorn	0.34+	ASGI server
sqlalchemy	2.0+	Async ORM
asyncpg	0.30+	PostgreSQL async driver
alembic	1.14+	Database migrations
google-genai	1.x	Google Gemini API client
llama-parse	0.6+	Document parsing service
pydantic	2.x	Data validation and serialisation
python-jose	3.3+	JWT decoding and verification
pillow	11.x	Image processing (thumbnails)
pymupdf	1.25+	PDF rasterisation
httpx	0.28+	Async HTTP client

Frontend Dependencies

Package	Version	Purpose
react	18.x	UI component library
typescript	5.x	Type-safe JavaScript
vite	5.x	Build tool and dev server
tailwindcss	3.x	Utility-first CSS framework
@azure/msal-browser	3.x	Azure AD authentication (browser)
@azure/msal-react	2.x	React bindings for MSAL
lucide-react	latest	Icon library
react-markdown	latest	Markdown rendering
recharts	2.x	Charting library (Analytics)