AdvanDEB Platform

Complete Architecture Rationale
All Platform Components and Integration

Version 12.24.2 - Architecture Revision December 2024

AdvanDEB Development Team

December 17, 2024

Abstract

This document provides comprehensive justification for every major architectural decision across the AdvanDEB Platform: authentication and user management, Knowledge Builder core functionality, Modeling Assistant design, integration architecture, and data model choices. This report explains the reasoning behind each design decision, evaluates alternative approaches, and documents the trade-offs that shaped the platform's architecture. This document serves as the authoritative reference for understanding why the platform is built the way it is, supporting future evolution and maintenance decisions.

Important Note: This document represents the current architectural design, not the final architecture of the system. The architecture is expected to evolve as the AdvanDEB project progresses through implementation, testing, and real-world usage. Design decisions documented here may be revised based on implementation challenges, user feedback, performance requirements, or new technical insights that emerge during development.

Contents

1 ARCHITECTURE REVISION - December 2024

Important: Architecture Model Updated

This document has been updated to reflect a significant architectural revision made in December 2024. The component roles and relationships have been clarified to better support implementation and user experience.

1.1 Revised Component Roles

The AdvanDEB Platform architecture has been revised to clarify the roles of each component:

advandeb-modeling-assistant is now the **Main Platform GUI** and single entry point for all users. It provides:

- Complete user interface (Vue.js frontend + FastAPI backend)
- Authentication and user management (Google OAuth)
- Role-based access control
- Chat interface, knowledge exploration, document ingestion, and modeling features
- Integration of all platform capabilities in one unified interface

advandeb-knowledge-builder is now a **Toolkit/Package** (not a standalone application). It provides:

- Python package with robust knowledge operations
- Fact extraction, stylized facts, document ingestion, graph building
- No user interface - pure backend logic
- Imported and used by Modeling Assistant backend
- Can be used independently for batch processing

advandeb-MCP is a **Tool Server** for LLM agent workflows. It provides:

- Model Context Protocol (MCP) server in Rust
- Exposes platform tools (both KB and MA operations) via MCP
- Internal service (no authentication) for agent-powered features
- Used by Modeling Assistant for chat and intelligent extraction
- Wraps Knowledge Builder operations as MCP tools

1.2 Key Changes from Previous Model

Previous Model:

- Knowledge Builder: Standalone FastAPI + Vue application
- Modeling Assistant: Separate FastAPI + Vue application
- Both had their own user interfaces and backends

New Model:

- Modeling Assistant: Main GUI with single frontend and backend
- Knowledge Builder: Toolkit package (no UI)
- MCP: Tool server exposing operations for LLM agents

1.3 Benefits of Revised Architecture

Simpler User Experience:

- One application to learn and use
- Single login with consistent interface
- Features shown/hidden based on user role
- No confusion about which component to access

Cleaner Architecture:

- Clear separation: GUI vs toolkit vs agent services
- Knowledge Builder becomes more reusable
- Easier to understand component responsibilities
- Better alignment with actual implementation

Easier Development and Deployment:

- Single frontend codebase to maintain
- KB toolkit can be versioned and tested independently
- One web application to deploy (MA)
- MCP developed in parallel without affecting GUI

1.4 Impact on This Document

Throughout this rationale document, references to "Knowledge Builder" and "Modeling Assistant" should be understood in the context of this revised architecture:

- When discussing user interfaces, authentication, and web features: refers to **Modeling Assistant GUI**
- When discussing knowledge operations, fact extraction, and graph building: refers to **Knowledge Builder toolkit**
- When discussing agent workflows and LLM integration: refers to **MCP server**
- When discussing integration: MA imports KB as package, MA calls MCP for agents

The fundamental architectural decisions (authentication, role model, database design, security) remain valid and are now primarily implemented in the Modeling Assistant, which serves as the platform's main interface.

1.5 Integration in New Model

Component Relationships:

- **Users** → authenticate with **Modeling Assistant GUI**
- **MA Backend** → imports **Knowledge Builder** as Python package
- **MA Backend** → calls **MCP Server** for agent features
- **MCP Server** → wraps **KB operations** as MCP tools
- All components share **MongoDB** database

Data Flow Example - Document Upload:

1. User logs into Modeling Assistant GUI

2. User uploads document via web interface
3. MA backend: `from advandeb_kb import ingest_document`
4. KB package processes and stores in MongoDB
5. MA returns success to frontend

Data Flow Example - AI Chat:

1. User types question in MA chat interface
2. MA sends request to MCP server
3. MCP executes tools (using KB functions) and calls Ollama
4. MCP returns formatted response to MA
5. MA displays response to user

For complete details on the revised architecture, see `ARCHITECTURE-REVISION.md` in the documentation repository.

2 Executive Summary

2.1 Purpose and Scope

This document provides the architectural rationale behind implementation choices. Every major architectural choice is examined through multiple lenses: the problem it solves, the alternatives that were considered, the trade-offs that were made, and the implications for the platform's evolution.

Comprehensive Coverage: This document covers all major platform components:

- **Architecture Revision (December 2024)**: Clarification of component roles - MA as main GUI, KB as toolkit, MCP as tool server
- **Authentication & User Management**: Google OAuth (hosted by MA), JWT tokens, capability-based roles, API keys, audit logging
- **Knowledge Builder Toolkit**: Fact extraction, stylized facts, document ingestion, knowledge graph building, agent workflows - provided as Python package
- **Modeling Assistant GUI**: Main platform interface, chat, knowledge exploration, modeling scenarios, role-based feature access
- **MCP Tool Server**: Model Context Protocol server exposing KB and MA operations for LLM agent workflows
- **Integration Architecture**: Package imports, internal service calls, shared database
- **Data Model**: MongoDB schema design, collection structure, relationship patterns, attribution mechanisms

Each decision reflects the realities of building a scientific research platform for 10-20 beta users scaling to approximately 100 users, prioritizing simplicity and rapid iteration over premature optimization.

2.2 Scientific Context: The AdvanDEB Research Project

The AdvanDEB Platform serves a specific scientific mission: advancing Dynamic Energy Budget (DEB) theory through transport network-based modeling (tDEB). Understanding this mission is essential for evaluating architectural decisions, as the platform's design directly supports the research methodology.

The Scientific Challenge: Dynamic Energy Budget models quantitatively describe energy acquisition and allocation in organisms, enabling connections across environmental and biological levels. These models have proven crucial for addressing societal challenges from food production and biodiversity conservation to ecosystem protection, resource utilization, and pollutant effects on organisms. However, modern DEB models have limitations—they store acquired energy immediately into reserves and distribute it using the "k-rule" in constant ratios to reproduction and other processes. While these models simulate long-term processes well, they struggle with short-term dynamics and fail to credibly connect state variables with organism physiology.

The tDEB Innovation: Incorporating transport networks (such as blood circulation) into DEB models addresses these shortcomings. Successful applications to whales and fish demonstrate high potential for broad applicability. However, many unresolved questions prevent widespread adoption of transport network-based DEB models (tDEB).

The AdvanDEB Project Goals: The project addresses these critical questions by: (i) verifying and ensuring consistency with known biological principles, (ii) defining clear rules for energy use during reproduction, (iii) testing generality and defining tDEB applicability, (iv) analyzing relationships between tDEB and other theories, and (v) developing formal assumption sets and unified terminology.

Novel Research Approach: AdvanDEB will leverage existing resources such as the DEB community's AmP database containing 4,500 species, using both conventional methods and AI-assisted techniques to identify applicability domains for tDEB, explore generalities within and across phylogenetic groups, and formalize tDEB theory. Additionally, AdvanDEB may unify DEB theory with Metabolic Theory of Ecology, creating potential for paradigm shifts in bioenergetics and opening new research pathways.

Platform Role in Research: The AdvanDEB Platform enables this scientific mission through novel approaches to literature review and knowledge base building. The platform must support: systematic extraction of biological facts from scientific literature, organization of heterogeneous knowledge about organisms across phylogenetic groups, linking knowledge to modeling parameters and assumptions, AI-assisted identification of patterns and generalities across species, and collaborative curation by domain experts with rigorous traceability.

The platform's architecture—particularly its three-tier knowledge model (Facts → Stylized Facts → Knowledge Graphs), AI agent framework for document processing, and MongoDB-based graph storage—directly supports these research requirements. Architectural decisions prioritize scientific rigor, knowledge traceability, and support for exploratory analysis over conventional software engineering metrics.

2.3 Platform Mission and Requirements

The AdvanDEB Platform exists at the intersection of scientific knowledge management and computational modeling. Its architecture must serve three fundamental requirements: scalable storage and querying of networked knowledge graphs spanning thousands of species and biological relationships, rigorous accountability and traceability required

by scientific research where every fact must be attributable to sources and curators, and flexible integration between knowledge management and modeling workflows that allows researchers to move fluidly between literature review, knowledge extraction, and model development.

The choice of MongoDB as the foundational database reflects these knowledge graph scalability requirements, while authentication, modeling, and integration architecture build upon this foundation to support the collaborative scientific process.

2.4 Architectural Philosophy

The architecture is guided by four foundational principles that emerged from careful analysis of the platform's core mission. First, scientific integrity takes precedence over convenience—every piece of knowledge must be traceable to its contributor, and every action must leave an audit trail. This principle manifests in authentication requirements, attribution systems, and immutable logging.

Second, the system favors collaboration over control. Rather than creating rigid hierarchies that force users into predefined boxes, the capability-based permission model allows users to grow their access as their needs evolve. A researcher might start as a basic curator, later request agent access for bulk extraction, and eventually earn reviewer status through demonstrated expertise. This organic growth pattern reflects how real research collaborations develop.

Third, simplicity emerges through integration rather than separation. While the platform comprises multiple components (Knowledge Builder and Modeling Assistant), they share authentication infrastructure, user models, and data conventions. Users experience a unified platform, not a collection of separate tools. This integration reduces friction while maintaining the benefits of component independence.

Fourth, security is implemented by default, not by configuration. Users cannot weaken authentication requirements, administrators cannot disable audit logging, and permissions are explicit rather than implicit. The system is designed to be secure even when operated by non-security-experts, because most scientific institutions lack dedicated security teams.

Note on Architectural Stability: While these principles guide all design decisions, the specific implementations documented here are not fixed. As the AdvanDEB project progresses through development and deployment, practical experience may reveal better approaches, uncover unforeseen constraints, or demonstrate that certain assumptions were incorrect. This document captures our current architectural thinking and will be updated to reflect significant changes as the platform evolves.

2.5 Document Purpose

This document answers "why these architectural patterns" and "what alternatives were rejected." It supports architectural reviews, technology evaluations, and future design decisions by providing detailed justification for each major choice.

Key architectural decisions documented include: the three-role capability model, the shared database architecture, the JWT token approach, the MongoDB choice for knowledge graph scalability, and the stateless design principles. Each choice involves trade-offs appropriate for beta scale (10-20 users) with growth capacity to approximately 100 users.

Architecture Evolution: The decisions documented here reflect our current understanding of requirements and constraints. As the AdvanDEB project advances through

implementation phases, we anticipate that practical experience will reveal optimization opportunities, uncover hidden constraints, or suggest alternative approaches. This document will be updated to reflect significant architectural changes, with version history tracking the evolution of design thinking as the platform matures.

2.6 Document Structure

Each section follows a consistent analytical pattern. We first state the design decision clearly, then explain the rationale behind it in narrative form. Alternative approaches are discussed with honest assessment of their pros and cons—rejected alternatives often had genuine merit but didn't fit our specific constraints. Trade-offs are examined explicitly because every architectural decision involves sacrificing something to gain something else. Finally, we discuss dependencies and implications, showing how each decision affects other parts of the system.

3 Scale Considerations and Design Constraints

3.1 Beta Platform: 10-20 Users, Maximum 100 Users

A fundamental driver of architectural decisions is the expected user scale. The AdvanDEB Platform targets 10-20 users during initial beta deployment, with maximum expected growth to approximately 100 users. This scale profoundly influences design choices in ways that differ from platforms targeting thousands or millions of users.

Implications for Architecture:

Database Design - At 100 users, the entire user database fits comfortably in memory. MongoDB performance concerns that dominate at scale (sharding strategies, replica set topologies, index optimization for millions of documents) are essentially irrelevant. Simple single-instance MongoDB with basic indexing provides sub-millisecond query times. This allows us to prioritize schema flexibility over query optimization.

Authentication Performance - With 100 users making perhaps 10-50 requests per minute each (1,000-5,000 requests/minute total), authentication overhead is negligible. The JWT validation approach provides excellent performance, but honestly, session-based authentication would also work fine at this scale. We choose JWTs primarily for architectural cleanliness and future-proofing, not performance necessity.

Rate Limiting - Rate limits exist primarily as safety guards against buggy scripts, not malicious abuse. With 10-20 trusted academic users, the risk profile is accidental errors rather than deliberate attacks. The generous limits (50-500 requests/minute per user) are unlikely to be hit during normal operation.

Operational Simplicity - Small user bases allow simpler operations. Manual approval of capability requests is feasible—administrators reviewing 1-2 requests per week is manageable, while 100 requests per day would demand automation. Audit log analysis can use basic MongoDB queries rather than requiring specialized analytics infrastructure. Backup and recovery procedures can be straightforward without complex disaster recovery orchestration.

Collaboration Focus - At this scale, every user can potentially know every other user. The platform can foster genuine research collaboration rather than anonymous mass contribution. Features like reviewer assignment can consider individual expertise rather than relying purely on algorithmic matching.

These scale characteristics allow us to defer certain architectural complexities that larger platforms require immediately: load balancing (single server suffices initially), database replication (backup is sufficient initially), caching layers (database performance is adequate), distributed tracing (simple logging works), and advanced monitoring (basic health checks suffice). We build these capabilities as modular additions when growth demands them, rather than over-engineering from the start.

4 System Architecture: The Fundamental Choices

4.1 The Three-Component Architecture (Revised December 2024)

Design Decision: Modeling Assistant as Main GUI, Knowledge Builder as Toolkit, MCP as Tool Server

The platform architecture has been revised to clarify component roles: Modeling Assistant serves as the main platform GUI and single user entry point, Knowledge Builder provides a toolkit of operations without UI, and MCP exposes platform tools for LLM agent workflows.

The architectural revision emerged from recognizing that users need **one unified interface** for all platform features, not separate applications for knowledge building versus modeling. The revised model positions:

Modeling Assistant as Main Platform GUI: All users authenticate through MA and access all features—chat, knowledge exploration, document ingestion, modeling scenarios—through one interface. Role-based access control determines which features each user sees. Users with curator roles access knowledge building features; users with modeling capabilities access scenario creation. This eliminates the confusion of "which application do I use" and provides seamless workflows where users move fluidly between knowledge exploration and modeling.

Knowledge Builder as Toolkit: The robust knowledge operations—fact extraction algorithms, stylized fact generation, document ingestion pipelines, knowledge graph building—are complex enough to warrant a separate repository, but they don't need their own user interface. Structuring KB as a Python package allows MA to import and use its functions while also enabling standalone batch processing. The toolkit can be versioned, tested, and evolved independently while being integrated into MA's user interface.

MCP as Tool Server: Exposing platform operations as MCP tools enables LLM agent workflows—intelligent fact extraction, conversational knowledge queries, scenario analysis. The Rust-based MCP server operates as an internal service (no authentication) that MA calls for agent-powered features. It wraps KB operations as tools and provides unified Ollama integration for LLM inference. This separation allows the computationally intensive agent operations to scale independently from the web tier.

The previous model—with separate KB and MA applications, each having their own UI—created unnecessary complexity. Users had to understand which component to access for which task. Shared features (authentication, knowledge browsing) existed in both UIs with potential inconsistencies. Development teams duplicated frontend code and design patterns. The revised model eliminates this duplication: one GUI (MA), one toolkit (KB), one agent service (MCP).

Integration Pattern: The revised architecture uses simple, direct integration:

- MA backend imports KB package: `from advandeb_kb import ingest_document`
- MA backend calls MCP server via internal HTTP/WebSocket for agent features
- MCP wraps KB operations as tools for LLM agents
- All components share MongoDB database for data consistency

This approach maintains the benefits of component separation (KB toolkit reusable, MCP scales independently, clear module boundaries) while providing the unified user experience that scientific researchers need. The integration overhead is minimal: Python package imports are native operations, and internal service calls have negligible latency.

The revised architecture explicitly optimizes for the actual user workflows: researchers don't think "now I'm doing knowledge building, now I'm doing modeling"—they fluidly explore knowledge, extract facts, build models, and iterate. The platform architecture now matches this mental model.

4.2 Shared Database Architecture

Design Decision: Single MongoDB instance for all platform data

Knowledge Builder and Modeling Assistant share one MongoDB database for all data: knowledge graphs, facts, documents, scenarios, models, users, roles, and audit logs.

The shared database architecture reflects MongoDB's selection as the platform's foundational data store, chosen primarily for networked knowledge graph and knowledge base scalability requirements. Once MongoDB is the database for knowledge management (the platform's core function), using it for user management, modeling data, and cross-component concerns provides strong consistency and operational simplicity.

The shared database provides immediate consistency across components. When a user creates a fact in KB, it's instantly queryable by MA for scenario building. When a scenario references a fact ID, MA queries the same database KB uses, eliminating synchronization delays. When a user's role changes, both components see the new permissions immediately. This strong consistency is possible because MongoDB ACID transactions span multiple collections within a database—user role update, audit log entry, and permission cache invalidation occur atomically.

Alternative architectures with separate databases per component would require complex synchronization. If KB and MA each had MongoDB instances, creating a fact in KB would need event streaming to replicate it to MA. During replication lag, MA users wouldn't see new facts. Conflict resolution would be needed if both components modified the same knowledge entity. User permission changes would require eventual consistency patterns with their inherent complexity.

A centralized authentication service with separate knowledge databases represents another alternative. Each component would have its own MongoDB for domain data, calling a shared auth API for user validation. This maintains component independence but fragments the knowledge graph—MA cannot directly query KB's knowledge collections; it must use HTTP APIs with latency overhead. More problematically, cross-cutting queries spanning knowledge and modeling data (find scenarios using facts created by user X) become expensive multi-hop API calls.

The shared database approach optimizes for the platform's primary use case: integrated knowledge management and modeling. Queries spanning components ("show me

scenarios built with these facts," "which facts support this model parameter") execute efficiently within one database. Transaction boundaries include both knowledge and user operations—creating a fact and logging the audit entry occur atomically. Backup and recovery are simpler with one database containing all platform state.

This decision explicitly couples KB and MA at the data layer. They share schema evolution, must coordinate database migrations, and share failure modes (database outage affects both). However, this coupling already exists logically—MA depends fundamentally on KB's knowledge for modeling. Making the coupling explicit through shared database simplifies rather than complicates: the components are tightly integrated by design, not independently deployable microservices.

Separate collections within the shared database provide logical separation: KB primarily works with facts, stylized_facts, documents, knowledge_graphs; MA primarily works with scenarios, models, results. The users, api_keys, and audit_logs collections serve both. This collection-level organization enables future database sharding if needed—knowledge collections could shard by taxonomic group (vertebrates vs. invertebrates) or organism type (aquatic vs. terrestrial) while user collections remain unsharded.

4.3 The advandeb-shared-utils Package

Design Decision: Python package containing shared authentication logic

Create a versioned Python package that both KB and MA backend services import for all authentication, authorization, and audit functionality.

The shared utilities package embodies the DRY (Don't Repeat Yourself) principle applied at the architecture level. JWT token generation logic is non-trivial—it involves cryptographic signing, claim formatting, expiration handling, and security considerations. Writing this code twice (once for KB, once for MA) doubles the testing burden, doubles the opportunity for bugs, and doubles the maintenance cost when security vulnerabilities are discovered. More insidiously, duplicate code tends to diverge over time as developers make changes in one location but forget the other, leading to subtle behavioral differences between components.

The package contains everything that must be identical across components: JWT generation and validation, Google OAuth client integration, Pydantic models for users and roles, permission checking functions (has_base_role, has_capability), API key hashing and validation, audit logging formatters, and MongoDB connection utilities. Code that might legitimately differ between components—business logic, API route handlers, component-specific models—deliberately stays outside the package.

This boundary is crucial. Shared code creates coupling, which is acceptable for cross-cutting concerns like authentication but harmful for business logic. If KB's document processing needs special capabilities, those belong in KB, not in shared-utils where they would create unnecessary dependencies for MA. The package provides primitives (check if user has capability X) not policies (curator may upload documents). Policies are composed from primitives in component-specific code.

Versioning the package allows controlled evolution. KB and MA can temporarily use different package versions during migrations, avoiding the big-bang upgrade problem. Semantic versioning communicates compatibility—patch versions fix bugs, minor versions add features backward-compatibly, major versions signal breaking changes. CI/CD

pipelines can automatically test both components against package updates before releasing them.

The alternative of duplicated code initially seems simpler—no package to publish, no version coordination, each team controls their own code. This simplicity is illusory. When a security audit reveals a vulnerability in JWT handling, fixes must be carefully replicated across codebases. When new capabilities are added, the permission checking logic must be updated in lockstep. The debugging nightmare when components behave slightly differently due to code drift far outweighs the overhead of maintaining a shared package.

5 Authentication Architecture

5.1 Google OAuth 2.0 as Primary Authentication

Design Decision: Use Google OAuth 2.0 for web UI authentication

Users authenticate to the platform via Google OAuth rather than traditional username/password credentials or other identity providers.

The choice of Google OAuth reflects a pragmatic assessment of our user base and institutional realities. The target audience for AdvanDEB consists primarily of academic researchers, most of whom already possess Google accounts through Gmail or Google Workspace. Asking these users to create yet another username and password creates friction that reduces adoption. Password fatigue is real—users reuse passwords across sites (security risk) or forget unique passwords (support burden). Google OAuth eliminates this friction entirely while delegating password security to an organization with vastly more security expertise than we possess.

From a security perspective, Google OAuth means we never see, store, or transmit user passwords. Our database cannot be compromised for passwords because we don't have them. Users who enable two-factor authentication in their Google account automatically bring that protection to AdvanDEB. Google's security team monitors for compromised credentials, unusual login patterns, and brute-force attacks—services we would struggle to implement ourselves. When Google discovers and patches authentication vulnerabilities, our users benefit automatically.

The credibility factor should not be underestimated. Academic institutions and grant funders increasingly scrutinize data security practices. "We use Google OAuth and never handle passwords" is far more convincing than "we've implemented password storage ourselves using bcrypt," regardless of the latter's technical adequacy. Institutional email addresses (researcher@university.edu) provide lightweight identity verification, and potential future integration with ORCID (the research identifier system) is easier through Google's OAuth framework.

We considered supporting multiple OAuth providers—GitHub for developers, ORCID for researchers, institutional SAML for universities. Each additional provider multiplies the testing and maintenance burden. Different providers return user data in different formats, requiring normalization logic. Users with multiple accounts at different providers need account linking workflows. These complexities might be justified for a platform targeting diverse audiences, but our user base is concentrated in academia where Google dominance is overwhelming.

Email/password authentication would provide complete control and independence from external services. However, it requires implementing secure password hashing (straightforward), password reset workflows (complex), email verification (requires email infrastructure), password strength enforcement (annoying users), and account lockout after failed attempts (support burden). More fundamentally, user experience suffers because researchers resist creating new passwords for domain-specific platforms.

The OAuth approach trades dependency on Google for simplicity and security. Google's hypothetical failure or policy changes could affect our platform, but this risk seems far lower than the security risks of home-grown authentication. Should circumstances change, the authentication abstraction allows adding alternative providers without disrupting existing users.

5.2 JWT Tokens for Session Management

Design Decision: Issue JWT tokens after OAuth authentication

After successful Google OAuth, the platform issues JWT access and refresh tokens rather than maintaining server-side sessions.

OAuth provides user identity, but JWT tokens provide session state. When a user successfully authenticates via Google, we gain knowledge of who they are but need a mechanism for that user to make subsequent requests without re-authenticating. Traditional web sessions store state on the server (typically in a database or Redis) and give the client a session ID cookie. Each request requires looking up the session to determine the user's identity and permissions.

JWT tokens invert this model by encoding the session state (user ID, role, capabilities) directly into a cryptographically signed token. The server generates the token with a private key, and any server with the corresponding public key can validate the token without database queries. Permission checks become pure CPU operations—parse token, verify signature, extract claims—completing in microseconds. This stateless architecture eliminates the session storage bottleneck that limits traditional systems.

The performance implications are substantial. A traditional session-based system queries the database or Redis on every authenticated request. At 100 requests per second, that's 100 database lookups just for authentication, before any business logic executes. JWT validation requires no I/O, making it essentially free compared to business logic costs. More importantly, this enables horizontal scaling without session affinity. Requests can be routed to any backend server without concern for which server holds the session state, because no server holds session state.

Security considerations around JWTs are often misunderstood. The common criticism that "JWTs can't be revoked" is overstated—proper JWT architecture mitigates this through short expiration times and refresh tokens. Our access tokens expire after one hour, limiting the window during which a stolen token remains valid. Refresh tokens, which last 30 days, are stored in the database and can be revoked immediately if compromise is detected. The user experience remains smooth because token refresh is automatic and invisible.

Statelessness also enables graceful handling of permission changes. When a user's role changes, the change takes effect within one hour (when their current access token expires and they refresh). For most use cases, this delay is acceptable. For urgent cases

(suspended user, security incident), administrators can force logout by revoking refresh tokens, requiring immediate re-authentication.

The token payload is base64-encoded JSON, readable by anyone who intercepts it. This is not a vulnerability if properly understood—sensitive data should never be placed in JWT claims. Our tokens contain only user ID, role, and capabilities, all of which the user already knows. For true secrets, separate encryption would be needed, but we have no such secrets in our authentication model.

Alternative approaches include server-side sessions (simple but doesn't scale), OAuth refresh tokens only (requires hitting Google on every request), or API session tokens (reinventing JWTs poorly). The JWT approach benefits from years of industry experience and well-tested libraries, making it the pragmatic choice for stateless authentication in distributed systems.

5.3 API Keys for Programmatic Access

Design Decision: Support long-lived API keys alongside JWT tokens

The platform provides API key authentication for scripts and automation, in addition to OAuth/JWT for interactive users.

Researchers frequently need programmatic access to platforms for bulk operations, data pipelines, and reproducible workflows. A biologist might write a Python script to upload 500 papers at once, or an ecologist might have a cron job that exports updated datasets nightly. OAuth workflows don't fit these scenarios because they require browser interaction to complete the authentication flow. API keys solve this by providing a secret token that can be embedded directly in scripts.

The design of our API key system learns from industry best practices established by platforms like GitHub and Stripe. Each key carries a distinguishable prefix (advk_) followed by randomly generated characters. The prefix enables quick identification in logs and accidental exposure detection (grep for "advk_" in public repositories). Only the hash of the key's secret portion is stored in our database, using SHA-256. When an incoming request presents an API key, we hash it and compare against stored hashes, exactly paralleling password verification.

This hashing approach means that compromising our database doesn't expose active keys. An attacker with database access sees hashes but cannot derive the original keys. Users see the plaintext key only once, immediately after generation, with explicit warnings to store it securely. This is admittedly less user-friendly than retrievable tokens but substantially more secure—there's no endpoint for attackers to call to "view my API keys."

API keys have independent lifecycle from user sessions. A researcher might use JWT tokens for interactive work in the web UI while also having API keys for their upload scripts. If the API key is compromised, it can be revoked without affecting their web access. Conversely, if they change their Google password (requiring re-authentication), their API keys continue working. This independence is crucial for automation—cron jobs shouldn't break because a researcher enabled 2FA on their Google account.

Each API key carries scopes automatically derived from the owning user's role and capabilities. A curator with analytics access receives API keys that can read and write knowledge plus export data. An explorator receives keys that can only read published

knowledge. These scopes are recalculated periodically (we could verify on each request, but caching for an hour balances security and performance). When a user's capabilities change, their API key permissions change accordingly.

Automatic expiration after 90 days forces periodic key rotation, limiting the window during which a compromised but undiscovered key remains valid. Users can set shorter expiration if they prefer (useful for temporary automation). The last-used timestamp helps users identify and delete abandoned keys that represent security risks.

Rate limiting is per-API-key rather than per-user, allowing fine-grained control. A researcher might have one key for bulk uploads (high rate limit, write permissions) and another for external collaborators (lower rate limit, read-only). This separation of concerns is impossible with single-token systems.

5.4 Single Sign-On Across Components

Design Decision: Same authentication tokens valid in KB and MA

Users authenticate once through Knowledge Builder and their tokens work seamlessly in Modeling Assistant without separate login.

Single sign-on represents a user experience decision with deep architectural implications. The natural workflow for AdvanDEB users involves moving fluidly between knowledge curation and model building. A researcher might upload papers in KB, use agents to extract data, then switch to MA to build scenarios using that data. Requiring reauthentication at the KB/MA boundary would create artificial friction that violates the platform's integrated vision.

Technically, SSO requires both components to trust the same JWT signing key and user database. When KB issues a JWT after OAuth, it signs the token with a secret key stored securely (environment variable, not in code). MA possesses the same secret key and can therefore validate KB-issued tokens. The token contains the user's ID, role, and capabilities—everything MA needs to enforce permissions without consulting KB.

This shared trust model couples KB and MA at the authentication layer. They must coordinate key rotation if the signing secret is compromised. They must agree on token format and claims structure. They must interpret capabilities identically. These coupling points are acceptable because authentication is an inherently shared concern—the notion of "different identity in KB versus MA" would fundamentally break the platform concept.

The alternative of separate authentication would require users to log in twice, probably with the same Google account, creating identical user records in both systems. This approach provides complete component independence but at severe cost to user experience. Users would need to track two sets of API keys. Permission changes would need to be synchronized. Audit trails would fragment across systems. None of these problems are insurmountable, but all of them work against the platform's goals.

Cookie domain configuration enables seamless browser transitions. If KB is hosted at kb.advandeb.org and MA at ma.advandeb.org, cookies can be set for the parent domain advandeb.org, making them accessible to both. The JWT access token typically lives in browser localStorage (accessible via JavaScript) while the refresh token lives in an HTTP-only cookie (protected from XSS). This split provides defense in depth—stealing the access token from localStorage grants only one hour of access.

Component-specific permissions are enforced locally using shared permission checking

logic from advandeb-shared-utils. MA verifies that scenario creation requires curator role, KB verifies that agent execution requires agent_access capability. Both use the same has_capability() function, ensuring consistent interpretation. The audit log records which component performed each action, enabling component-specific security analysis while maintaining unified user identity.

This SSO architecture assumes that KB and MA have similar security requirements. If future expansion adds a component requiring higher security (perhaps a clinical data module), that component might demand separate authentication with additional factors. The architecture accommodates this through capability requirements—the clinical module could require a capability that's only granted after additional verification. The baseline SSO serves the common case while allowing exceptions for special cases.

6 Role Model: Capability-Based Permissions

6.1 Evolution Through Three Versions

Design Decision: Base roles with optional capabilities

The architecture uses three base roles (Administrator, Knowledge Curator, Knowledge Explorator) with optional capabilities (Knowledge Creation, Agent Access, Analytics Access, Reviewer Status) that curators can request as needed.

The capability model architecture distinguishes between *who you are* (your base access level to the platform) and *what you can do* (your permissions for specific operations). This separation addresses a fundamental challenge in permission design: researchers need different capabilities at different times in their work, but their core identity and access level remains stable.

A researcher's identity as a curator is stable, but their need for direct creation privileges or agent access varies over time. A new user starts with suggestion-based workflow—they upload documents and suggest facts or graph nodes for review. This lightweight participation builds familiarity while maintaining quality control. After demonstrating understanding and providing valuable suggestions, they might request Knowledge Creation capability to create content directly (still subject to review but with faster workflow). Still later, they might request agent access to scale their work, or reviewer status to evaluate others' contributions. This graduated progression reflects natural workflow evolution rather than forcing users into rigid categories.

The architecture implements this through base roles plus optional capabilities. The three base roles reflect genuine differences in platform access level: Administrator (operates the platform), Knowledge Curator (uploads documents and suggests knowledge contributions), Knowledge Explorator (reads content). Base curators can upload documents and suggest facts or graph nodes—the core participation workflow. Capabilities represent additional permissions that curators request as needed: Knowledge Creation (create facts/nodes directly), Agent Access (run AI agents), Analytics Access (bulk data export), Reviewer Status (approve submissions from other curators). This model provides both simplicity (three base roles cover all users) and flexibility (capabilities combine freely to match individual workflows).

The key insight is recognizing that researchers naturally wear multiple hats and grow into responsibilities. The same biologist might start by uploading papers and suggesting

facts (base curator activity), later earn direct creation rights to work more efficiently (Knowledge Creation capability), run extraction agents to process literature at scale (Agent Access), and export aggregated data for analysis (Analytics Access), all within a single research session. Rather than forcing them to switch between role identities or be permanently locked into a single role, the capability model allows their permissions to grow organically with their needs and demonstrated trust.

The capability request workflow embeds learning and trust-building into permission evolution. New users start with suggestion-based curator access, learn the system, and propose knowledge contributions. When they've proven their understanding, they request Knowledge Creation capability with examples of their quality suggestions. Administrators review the request, considering the user's track record. This graduated access model aligns security (capabilities granted based on demonstrated trust) with usability (users aren't overwhelmed with permissions they don't yet need or understand).

This approach accepts that some capabilities are common stepping stones while others are specialized. Most curators will progress from suggestions to direct creation as they demonstrate competence. Many will eventually request reviewer status because reviewing is a natural evolution of contributing. Fewer will need analytics access because bulk data export serves specialized use cases. The model adapts naturally to this distribution—common capabilities are approved routinely based on demonstrated contribution while unusual requests trigger scrutiny.

6.2 The Four Capabilities

Design Decision: Knowledge Creation, Agent Access, Analytics Access, and Reviewer Status as separately requestable capabilities

These specific four capabilities represent meaningful permission boundaries rather than arbitrary divisions.

Each capability corresponds to a genuine security or quality concern that justifies treating it as a separable permission.

Knowledge Creation is the first capability most curators request, yet it represents an important quality control boundary. Base curators submit suggestions—lightweight proposals for facts or graph nodes that require reviewer approval before entering the system. This suggestion workflow has low risk: poor suggestions can be quickly rejected without polluting the knowledge base. Knowledge Creation capability allows curators to create facts and graph nodes directly (still subject to review, but with higher priority and faster workflow). This distinction matters because direct creation, even with review, carries more weight than suggestions. A fact created by a trusted curator is presumed higher quality than an anonymous suggestion, affecting how reviewers allocate attention. The capability isn't granted automatically because quality control depends on knowing who has direct creation rights.

The progression from suggestion to creation mirrors academic advancement: students propose ideas under supervision before gaining independence to assert claims. New curators learn the platform's knowledge model by suggesting facts and receiving feedback. After demonstrating understanding through quality suggestions, they earn creation rights to work more efficiently. This graduated access prevents knowledge base pollution while welcoming broad participation.

Agent Access controls the ability to run AI agents that make external API calls (to LLM providers, for example) and consume system quotas. Misbehaving agents could rack up API costs or degrade system performance. Not every curator needs this power—many work effectively by manually creating knowledge entries. Those who do need agents (for bulk extraction from many papers) can request access with justification that allows administrators to assess the request's legitimacy.

The capability exists separately because the risk profile differs from basic curation. A curator creating facts manually can do damage only at human speed—entering wrong data, submitting low-quality content. An agent running in a loop can cause havoc at machine speed—thousands of API calls, database writes overwhelming infrastructure, quotas exhausted. This risk distinction justifies the approval gate.

Analytics Access controls bulk data export and API key generation. Curators can export individual facts or small datasets for their own use without this capability. Analytics Access enables exporting the entire knowledge base, generating API keys with broad read permissions, and running complex queries that might impact database performance. This capability exists because bulk export enables data exfiltration—a malicious user could download everything and republish it elsewhere. While much knowledge is public, aggregating it represents value that users might want to protect.

The capability isn't needed for normal research workflows. A biologist modeling fish populations needs to query relevant species data, not export the entire database. Analytics Access serves researchers doing meta-analysis across many domains, data scientists training ML models on the knowledge corpus, or institutional partners wanting to maintain local copies. These legitimate use cases justify granting the capability, but they represent a small fraction of users.

Reviewer Status controls access to the review queue and the ability to approve or reject submitted knowledge and suggestions. Quality control is crucial for maintaining platform credibility—poor reviews admit bad content while excessive rejection discourages contributors. Not every experienced curator should be a reviewer; some are domain experts comfortable contributing knowledge but not comfortable judging others' work. Conversely, some expert reviewers might contribute little themselves while providing valuable quality control.

Treating review as a capability rather than a separate role acknowledges that reviewing and curating are related but distinct activities. The same person might submit a fact (as a curator) and later review someone else's fact (with reviewer status). This dual activity pattern is natural in scientific collaboration—domain experts both contribute and evaluate contributions.

These four capabilities emerged from analysis of permission patterns and security requirements. We examined what curators do and identified permission boundaries with security or quality implications worth protecting. The number is four because that's how many meaningful boundaries exist: direct creation vs. suggestion (quality control progression), agent operations (cost and performance risk), bulk data access (exfiltration risk), and quality control (content integrity risk). Each capability addresses a distinct concern that justifies the approval workflow.

6.3 Why Knowledge Explorator Matters

Design Decision: Provide read-only role for non-contributing users

The platform offers a role for users who browse knowledge without contributing, rather than requiring all users to be potential contributors.

The Knowledge Explorator role reflects recognition that not all valuable platform users contribute content. Students learning about a domain, journalists researching articles, policymakers gathering information, and interested public members exploring science might never upload a paper or create a fact. If curator is the minimum role required, these users are denied access or forced to request permissions they don't need and won't use.

Providing a read-only role lowers the barrier to platform engagement. Approval is straightforward because explorators can't damage data or consume significant resources. An administrator can approve explorator requests quickly, perhaps automatically, knowing the risk is minimal. Some explorators will eventually become curators as they gain confidence and identify gaps they could fill. This natural progression from consumer to contributor is common in online communities; blocking it with unnecessary permission gates would hurt community growth.

The role also serves institutional use cases. A university might want all biology graduate students to access the platform as part of coursework without those students needing curator permissions. A research institute might provide explorator access to administrative staff who need to look up information but shouldn't contribute. These use cases require a lightweight permission level that doesn't carry the responsibilities of curation.

From a scaling perspective, explorators generate revenue-negative load (consumption without contribution) but this is acceptable in service of platform mission. Scientific knowledge sharing benefits from broad access, and the computational cost of serving read queries is low. Restricting access to contributors only would create a closed community that contradicts the open science values motivating the platform.

Explorators have capabilities like saving searches, creating private annotations, and exporting small datasets for personal use. These features enable productive use without risking the shared knowledge base. The private annotations feature specifically supports the learning use case—students can take notes on concepts without those notes polluting the public knowledge space.

The distinction between explorator and curator matters fundamentally: curators can suggest changes to the knowledge base by submitting facts for review, while explorators have read-only access. Even without optional capabilities, base curator role enables the core contribution workflow—uploading documents, creating facts, submitting knowledge for review. Explorator communicates "this account is for reading and learning" while curator communicates "this account can contribute to the knowledge base." The distinction is not about waiting for permissions but about intended use: consumption versus contribution.

7 Database Design: MongoDB as Foundation

7.1 Why MongoDB Over Relational Databases

Design Decision: Use MongoDB as the primary database for the entire platform

MongoDB was chosen as the foundational database technology for AdvanDEB, primarily driven by knowledge graph and knowledge base requirements, with user management built on the same infrastructure for operational simplicity.

The choice of MongoDB is fundamentally driven by the platform's core mission: managing a networked knowledge graph and scalable knowledge base for bioenergetics research. Knowledge graphs consist of heterogeneous nodes (species, physiological processes, metabolic pathways, bioenergetic parameters) with varying properties and relationships. Facts extracted from literature have different structures depending on content. Stylized facts synthesize varying numbers of source facts. This inherent heterogeneity makes rigid relational schemas problematic—every new entity type or relationship pattern would require schema migrations.

MongoDB's document model naturally represents graph structures: nodes are documents with flexible properties, edges are embedded arrays or references. A species node might have {taxonomy, habitat, metabolic_rate, reproduction_parameters} while a physiological process node has {energy_allocation, transport_mechanisms, temporal_dynamics}. Both coexist in the same collection with different schemas. Graph queries like "find all nodes connected to node X within 2 hops" translate to MongoDB aggregation pipelines, while the same query in relational databases requires recursive CTEs or multiple self-joins that perform poorly at scale.

The scalability requirements for networked knowledge graphs strongly favor document databases. As the knowledge base grows to thousands of interconnected entities, MongoDB's horizontal scaling through sharding enables distributing graph partitions across servers. Graph queries remain efficient because MongoDB can parallelize across shards. PostgreSQL with graph extensions (like AGE) provides graph capabilities but sharding is complex and less mature. Dedicated graph databases (Neo4j, ArangoDB) excel at graph queries but add operational complexity—running two database systems (graph for knowledge, relational for users) multiplies overhead.

Once MongoDB is chosen for knowledge graph storage (the platform's primary data), using it for user management provides operational consistency rather than introducing a second database technology. User metadata actually benefits from MongoDB's flexibility: affiliation structures vary across institutions, research areas use different taxonomies, and future features (ORCID integration, institutional roles, publication records) can be added without migrations.

The graph-oriented use case shapes specific MongoDB features used: compound indexes on node connections (source_id + edge_type + target_id) enable fast graph traversal, aggregation pipelines implement graph algorithms (shortest paths, community detection), and flexible schema allows evolving graph ontologies without downtime. These graph-centric features distinguish MongoDB from traditional document databases optimized for CRUD operations.

The capabilities array in user documents illustrates MongoDB's secondary benefits for user management. In a relational model, capabilities would require a junction table

with JOINs for permission checks. MongoDB stores capabilities as arrays, making checks trivial and updates simple. However, this user management flexibility is a convenient side effect, not the primary driver—the platform would use MongoDB even if user management was more complex, because the knowledge graph requirements are paramount.

The JSON-native storage provides seamless data flow: knowledge extracted from documents is JSON, MongoDB stores it as BSON, HTTP APIs return JSON. This alignment extends to user data, but more importantly to knowledge entities where complex nested structures (graph nodes with properties, facts with entity arrays, stylized facts with evidence references) map naturally to nested JSON without ORM impedance mismatches.

Operational simplicity from single-database architecture matters more as data volumes grow. The platform's primary data—the knowledge graph—will scale to gigabytes or terabytes (thousands of papers, millions of facts, complex interconnected graphs). MongoDB expertise, backup procedures, query optimization, and monitoring infrastructure developed for knowledge storage automatically apply to user data. Adding PostgreSQL would split operational focus without addressing the core challenge: scalable networked knowledge graph storage.

Consistency guarantees in MongoDB have improved dramatically with multi-document transactions (introduced in MongoDB 4.0, mature by 4.4). Creating a fact and linking it to a knowledge graph node can occur atomically. While PostgreSQL's transactions are more mature, MongoDB's capabilities suffice for knowledge management operations. The platform's transaction patterns (create knowledge entity + update relationships) fit MongoDB's transaction model well.

The decision explicitly acknowledges MongoDB's weaknesses for certain query patterns. Complex analytical queries spanning many collections with intricate JOINs would be easier in PostgreSQL. However, the platform's query patterns favor document and graph operations: "find facts matching text query," "traverse graph from node X," "retrieve stylized fact with all source facts," "find entities related through path." These patterns leverage MongoDB's strengths (text search, aggregation pipelines, denormalized storage) rather than its weaknesses (complex JOINs).

Alternative NoSQL databases were evaluated against knowledge graph requirements:

- **Neo4j**: Superior graph queries but adds database diversity, requires learning Cypher, and sharding is complex
- **ArangoDB**: Multi-model (document + graph) but less mature ecosystem and smaller community
- **Cassandra**: Optimized for writes and time-series but graph queries are difficult
- **DynamoDB**: Managed NoSQL but graph queries require complex access patterns and costs scale unpredictably

MongoDB balances graph capabilities (good enough), document flexibility (excellent), operational maturity (production-proven), and community support (extensive). For a small team building a research platform, MongoDB's "good at many things" profile outweighs specialized databases' "excellent at one thing" profiles.

7.2 Collection Design and Data Modeling

Design Decision: Four collections for user data—users, capability_requests, api_keys, audit_logs

User-related data is organized into these four collections rather than a single collection or normalized relational schema.

The users collection represents the current state of user identity and permissions. Each document contains all information needed to authenticate and authorize a user: their Google ID (immutable, used for lookup), email and name (from Google, may change), role and capabilities (platform-controlled), status (active/suspended/pending), and metadata about their affiliation and research interests. Embedding metadata directly in the user document rather than normalizing to a separate table reflects MongoDB best practices—data that's always queried together should live together.

The decision to store capabilities as an array in the user document deserves particular explanation because it conflicts with relational normalization. In third normal form, a many-to-many relationship (users have many capabilities, capabilities belong to many users) would use a junction table. This normalized approach optimizes for data integrity—adding a new capability type doesn't require touching existing user records. However, it pessimizes the common case of permission checks during request processing.

Every authenticated request checks the user's permissions. With the array model, this check happens entirely in memory after fetching the user document—no additional queries. With the junction table model, each permission check requires either a JOIN (expensive) or a separate query to the junction table (network overhead). At hundreds of requests per second, this overhead accumulates significantly.

The trade-off is that adding new capability types requires updating all user documents that should have it. This is acceptable because capability types are added rarely (perhaps a few times per year) while permission checks occur millions of times per day. MongoDB optimizes the common case at the expense of the rare case.

The capability_requests collection maintains historical records of all permission requests—both for initial base roles and for additional capabilities. This data doesn't belong in the users collection because each user might have many requests over time, and requests contain reviewer notes and timestamps irrelevant to authentication. Separating concerns keeps the users collection fast for the hot path (authentication) while the requests collection supports the slower path (administration and auditing).

The api_keys collection exists separately because users can have multiple API keys, each with independent expiration, rate limits, and usage tracking. The last_used_at timestamp enables users to identify and delete abandoned keys. The key_prefix supports fast lookups without full-text search. Storing only the SHA-256 hash of the key protects against database compromise. None of this data is needed during web authentication, so polluting the users collection with it would slow down the common case to serve a specialized case.

The audit_logs collection grows continuously and has different access patterns than other collections. Logs are append-only (never updated after writing), queried by time range, filtered by user or action or component, and retained longer than other data. These characteristics justify a separate collection with specialized indexes. As the collection grows large, it can be partitioned by date or moved to archival storage without affecting operational data. Mixing audit logs with users would eventually degrade user lookup

performance as logs accumulate.

This four-collection design balances normalization (don't duplicate data unnecessarily) with denormalization (duplicate data when it improves performance). The user document denormalizes by embedding metadata and capabilities because permission checks need this data. Capability requests and API keys normalize because they're queried separately and have different lifecycles. Audit logs separate because their access patterns differ fundamentally. These decisions emerge from understanding the workload, not from applying rigid rules about normalization.

8 Security Architecture

8.1 Defense in Depth Through Multiple Auth Methods

Design Decision: Support three auth methods—OAuth, JWT, API keys

The platform allows authentication through Google OAuth, JWT tokens, or API keys rather than forcing a single method.

Supporting multiple authentication methods reflects recognition that different use cases have different requirements and risk profiles. Web browser users authenticate via OAuth because it provides the best user experience—no password to manage, security delegated to Google, 2FA handled automatically. Once authenticated, they receive JWT tokens for subsequent API calls because tokens enable stateless authentication at scale. Script and automation users receive API keys because OAuth requires browser interaction that doesn't work for headless operation.

This diversity provides defense in depth. If OAuth is temporarily unavailable due to Google outage, users with API keys can continue programmatic access. If a user's JWT refresh token is stolen, revoking it doesn't invalidate their separately issued API keys (assuming the keys aren't also compromised). If vulnerability is discovered in the JWT library, API keys provide a fallback while patches are applied. No single point of failure exists because multiple independent authentication paths serve the platform.

Each method has limitations that the others address. OAuth depends on Google's availability and policies. JWTs can't be immediately revoked before expiration. API keys require secure storage that many users handle poorly. Combining methods allows users to choose the approach matching their security posture and use case while providing the platform with fallback options.

The implementation shares authorization logic across methods through advandeb-shared-utils. Whether a user authenticates with OAuth+JWT or API key, the same permission checking functions determine their access. This consistency prevents security gaps where one path accidentally bypasses checks that another path enforces. The authentication method is recorded in audit logs, enabling analysis of whether different methods show different security incident rates.

8.2 Rate Limiting for Abuse Prevention

Design Decision: Per-user rate limits based on capabilities, implemented with Redis

Requests are limited based on who made them and what capabilities they have, using Redis for tracking.

Rate limiting serves multiple purposes that justify its implementation complexity. First, it prevents accidental denial-of-service from buggy scripts. A researcher's Python script with an infinite loop shouldn't be able to exhaust database connections or CPU time. Second, it limits intentional abuse from malicious users attempting to scrape data or overwhelm infrastructure. Third, it provides fair resource allocation—users with higher capabilities (and typically greater need) get higher limits while preventing any single user from monopolizing resources.

The capability-based limits reflect the reality that different user types have legitimately different needs. A Knowledge Explorator browsing the platform might make 50 requests per minute clicking through facts and papers. A Knowledge Curator actively contributing might make 100 requests per minute as they upload documents and create entries. A Curator with Analytics Access running data export scripts might legitimately need 500 requests per minute. For our beta user base (10-20 users initially), these limits are generous and unlikely to be hit during normal operation. They primarily serve as safety guards against accidental infinite loops in scripts or buggy automation, rather than defending against malicious abuse.

Redis implements rate limiting because the check must occur on every request with minimal latency. Storing rate limit counters in MongoDB would add database load and latency that defeats the purpose. Redis keeps counters in memory for microsecond access times and supports atomic increment operations that prevent race conditions in distributed systems. The built-in TTL (time-to-live) feature automatically expires old counters without manual cleanup, implementing sliding window rate limits efficiently.

The specific algorithm used is token bucket, where each user has a bucket holding tokens and each request consumes a token. Tokens refill at a constant rate (their rate limit). This allows brief bursts above the average rate while preventing sustained high request rates. A user with 200 req/min limit can make 300 requests in a single minute if they made few requests in prior minutes, but cannot sustain 300 req/min continuously. This flexibility accommodates realistic usage patterns (periods of intense activity followed by quiet periods) while preventing abuse.

Alternative approaches like database-based limiting (too slow), proxy-level limiting (doesn't know about user capabilities), or no limiting (invites abuse) were rejected for failing to balance the competing concerns of performance, fairness, and flexibility.

8.3 Comprehensive Audit Logging

Design Decision: Log all writes, auth events, and permission changes

The platform logs every operation that modifies state or changes security posture, storing logs in audit_logs collection.

Audit logging serves scientific integrity, security monitoring, regulatory compliance,

and user support needs that collectively justify its operational cost. For scientific integrity, every knowledge contribution must be attributable to its creator. If a questionable fact appears in the database, audit logs show who created it, when, and from what IP address. This traceability builds trust—users contributing good-faith knowledge aren't tainted by bad actors because contributions are individually traceable.

Security monitoring requires logging to detect compromised accounts exhibiting unusual behavior. A curator who typically creates 10 facts per day suddenly creating 1000 per hour suggests account compromise. A user accessing the platform from a new country deserves verification. These patterns only emerge from analyzing logged behavior over time. Real-time monitoring can trigger alerts for administrators to investigate, while historical logs support forensic analysis after incident discovery.

Compliance requirements from funding agencies, institutional review boards, and data protection regulations often mandate audit trails. GDPR requires tracking who accessed personal data and when. Grant funders want evidence that research data was handled according to protocols. Future certifications (HIPAA if health data is added, FedRAMP for government use) require comprehensive logging. Building audit capabilities from the start is far easier than retrofitting them later when certification is needed.

User support benefits from audit logs when answering "what happened to my data?" questions. A user reports that their draft fact disappeared—logs show they accidentally deleted it yesterday. A user can't access a feature they swear they had permission for—logs show their capabilities were revoked last week with documented justification. These mundane support scenarios resolve quickly with good logs but become frustrating mysteries without them.

The decision to log writes but not reads reflects cost/benefit analysis. Write operations are relatively rare and high-impact. A fact created incorrectly might mislead researchers. An API key generated carelessly might enable data exfiltration. These operations justify the storage and performance cost of logging. Read operations are far more frequent (perhaps 100x more than writes) and lower impact. Logging every read would generate massive log volumes with limited security value—reading public data isn't typically a security event. For our beta deployment with 10-20 users, log volume remains manageable even with comprehensive write logging, allowing detailed forensic analysis without storage concerns.

The exception to read-logging is failed authorization attempts. If a user tries to access something they lack permission for, that attempt is logged. This detects permission confusion (user expects access they don't have, suggesting UX problems) and potential attackers probing for vulnerabilities. Successful reads of public data aren't logged, but unauthorized attempts are, providing security visibility without drowning in noise.

Logs are immutable after writing—entries can't be updated or deleted. This prevents evidence tampering where a compromised account tries to erase its malicious actions. Log immutability requires careful design to avoid logging sensitive data (never log passwords, even hashed ones; never log full API keys, only prefixes). If sensitive data is accidentally logged, the entire log entry must be rotated out of accessibility rather than edited.

9 Scalability and Performance

9.1 Stateless Architecture for Horizontal Scaling

Design Decision: Store all session state in JWT tokens, not on servers

Authentication state lives in cryptographically signed tokens rather than server-side session stores.

The stateless architecture emerged from recognizing that session storage becomes the bottleneck in traditional web applications at scale. Each request in a session-based system queries the session store (database or Redis) to retrieve user identity and permissions. At 1000 requests per second, that's 1000 session lookups per second before any business logic executes. This throughput ceiling appears regardless of how many application servers you add because they all contend for the same session store.

Stateless JWT authentication eliminates this bottleneck by encoding session state in the token itself. The user document is fetched once during authentication and its essential fields (user ID, role, capabilities) are encoded into a JWT signed with a secret key. Subsequent requests present this token, and any application server can validate it using the secret key without database queries. Validation requires only CPU time (cryptographic signature verification) which is plentiful compared to I/O time.

This architecture enables horizontal scaling as user count grows. For our initial beta deployment (10-20 users), a single application server suffices. As we approach 100 users, additional servers can be added trivially without session affinity concerns. The load balancer routes requests randomly because no server maintains session state. This design principle—stateless from the start—means scaling requires no architectural changes, only resource addition.

The stateless model trades immediate revocation capability for scalability. A traditional session can be deleted from the session store, immediately terminating access. A JWT remains valid until expiration even if the user is suspended. This is mitigated by short token lifetimes (1 hour for access tokens) and revocable refresh tokens. Suspending a user blocks refresh, so they lose access within an hour maximum. For urgent cases, administrators can add the token's JTI (unique identifier) to a revocation list, though this reintroduces a database lookup.

The approach assumes that user permissions change infrequently relative to request rate. A user making 100 requests per minute with permissions changing once per day sees massive benefit from stateless checking. If permissions changed every minute, the caching inherent in JWT tokens would become problematic. This assumption holds for our use case—role changes are administrative actions occurring occasionally, not continuous events.

9.2 Token Lifetime Balancing Security and Convenience

Design Decision: 1-hour access tokens, 30-day refresh tokens

Access tokens expire quickly while refresh tokens persist longer, balancing security risk with user convenience.

The dual-token approach with different lifetimes reflects a fundamental tension in

authentication system design. Users want persistent sessions—logging in once and staying logged in indefinitely. Security requires short-lived credentials—stolen tokens should stop working quickly. The dual-token pattern reconciles these through separation of concerns: access tokens for authorization (short-lived, used frequently) and refresh tokens for re-authentication (longer-lived, used rarely).

Access tokens authorize individual requests. They're included in every API call, visible in request logs, potentially exposed to browser JavaScript. This exposure surface area justifies short lifetimes. If an access token is stolen through XSS attack or compromised log file, the attacker gains access only until expiration—one hour. This limits damage compared to tokens that work indefinitely. The one-hour lifetime specifically balances typical session duration (users actively working stay under an hour between requests) with security exposure window.

Refresh tokens enable obtaining new access tokens without re-authenticating. They're stored in HTTP-only cookies (inaccessible to JavaScript, reducing XSS risk) and used only when access tokens expire. A user making 100 requests per hour uses their access token 100 times but their refresh token only once. This reduced exposure justifies longer lifetime—30 days means users stay logged in for a month, matching typical "remember me" functionality.

The 30-day refresh lifetime emerged from user behavior analysis. Researchers working daily use the platform continuously and should stay logged in. Researchers taking vacation or working on other projects might not access the platform for weeks. Automatically logging out monthly balances convenience (don't interrupt active users) with security (don't leave abandoned accounts logged in indefinitely). Users who need shorter sessions for security reasons can log out manually.

Refresh tokens are revocable despite being longer-lived. They're stored in the database, so suspending a user deletes their refresh tokens. Next time they try to refresh their access token, the attempt fails and they must re-authenticate. This provides administrators with an immediate logout capability for security incidents without giving up the performance benefits of stateless access tokens.

The two-token pattern follows industry standard established by OAuth 2.0 and adopted by major platforms. GitHub, Google, and AWS all use similar patterns. This is not innovation—it's applying well-understood solutions to standard problems. The specific lifetime values (1 hour and 30 days) are parameters we can adjust based on observed usage patterns and security incident analysis.

10 Deployment and Operations

10.1 Three-Environment Strategy

Design Decision: Separate development, staging, and production environments

The platform deploys to three distinct environments with different configuration rather than a single environment or ad-hoc deployment.

The three-environment strategy reflects lessons learned from decades of software deployment experience. Development environments run on developer machines with minimal infrastructure—local MongoDB, mock email, no rate limiting. This configuration

optimizes for fast iteration. Developers can test changes in seconds, debug with full access, and experiment with risky changes that would be inappropriate in production. The environment's isolation means developer mistakes affect only their own machine.

Staging environments replicate production configuration as closely as practical. They use hosted MongoDB, real email services, production-like rate limits, and actual Google OAuth (configured for a test domain). This environment catches integration issues that local development misses—network latency, external service timeouts, configuration errors, load behavior. Client acceptance testing occurs in staging, allowing stakeholders to preview features before production deployment. The principle of "test in production-like conditions" is achieved without risking production.

Production environments add redundancy and monitoring beyond staging. MongoDB runs as a replica set for high availability—primary node failures promote a secondary automatically. Redis runs as a cluster for rate limiting availability. Multiple backend servers sit behind load balancers. SSL certificates enable HTTPS. Monitoring dashboards track error rates, response times, and resource usage. Alerts notify operators of anomalies. Automated backups run continuously with tested restore procedures.

This environment progression provides safety nets. A bug that passes unit tests might be caught in development integration testing. A bug that passes development might be caught in staging during acceptance tests. A bug that passes staging might still be caught by production monitoring before affecting many users. No single check prevents all issues, but layers of checking reduce failure probability dramatically.

The alternative of deploying directly to production or using only dev/prod without staging increases risk unacceptably. Development environments differ too much from production (localhost vs hosted, mock vs real external services) to catch integration issues. Production deployments without staging preview create all-or-nothing scenarios where bugs affect users before they're discovered. The staging environment specifically exists to bridge the gap between development idealism and production realism.

Configuration management across environments uses environment variables and configuration files, never hardcoded values. The same code artifact (Docker container) deploys to all three environments with different configuration. This ensures that staging tests are testing the actual code that will deploy to production, not a similar-but-different version. Configuration-as-code practices (Terraform for infrastructure, Ansible for configuration) enable reproducible environment creation and disaster recovery.

11 Knowledge Builder Architecture

The Knowledge Builder is a semi-supervised system for constructing the platform's knowledge base through both automated and manual processes. At its core, it enables chatbot-mediated interaction between curators, documents, and existing knowledge. Rather than requiring purely manual extraction or relying entirely on black-box automation, the system combines AI assistance with human oversight: curators can converse with documents through a chatbot interface that suggests facts to extract, references existing knowledge to maintain consistency, and allows manual refinement before publication. This hybrid approach treats knowledge building as an interactive dialog—the curator asks "What does this paper say about metabolic scaling?" and the system extracts candidate facts while showing how they relate to existing knowledge. The curator reviews, refines, and approves, ensuring quality while leveraging automation for speed. The result is a supervised automation workflow where humans guide and validate while AI handles extraction and

initial structuring.

11.1 Knowledge Representation: Facts, Stylized Facts, and Graphs

Design Decision: Three-tier knowledge representation model

Knowledge is represented at three distinct abstraction levels: Facts (raw observations), Stylized Facts (synthesized patterns), and Knowledge Graphs (structured relationships).

The three-tier model reflects how scientific knowledge naturally evolves from specific observations to general principles. Facts represent atomic observations extracted from literature or data—"Species X allocates 40% of energy to reproduction at temperature T" or "Organism Y shows maximum ingestion rate R at body mass M." These facts are numerous, specific, and tied directly to source materials for traceability.

Stylized Facts aggregate multiple related facts into higher-level statements—"Species X exhibits temperature-dependent energy allocation patterns" or "Phylogenetic group Y shows allometric scaling of metabolic rate with exponent 0.75." A single stylized fact might synthesize evidence from dozens of facts, providing compression and clarity while maintaining traceability through fact_ids references. This intermediate layer serves modeling needs: modelers need patterns and generalizations, not raw observations, but they also need to drill down to evidence when assumptions are questioned.

Knowledge Graphs provide structural representation of entities and relationships. While facts and stylized facts use natural language (flexible but ambiguous), graphs use formal node-edge structures (rigid but queryable). The same biological knowledge might exist as facts ("Paper reports Species A preys on Species B"), stylized facts ("Predator-prey dynamics between A and B show Type II functional response"), and graph edges (A –[preys_on]–¿ B). This redundancy is intentional—different use cases favor different representations.

Typed Relationships Between Knowledge Entities: The knowledge graph architecture supports rich relationship types between facts, stylized facts, and other knowledge entities, enabling explicit representation of epistemic relationships:

- **Supporting Evidence** (supports): Fact B provides evidence supporting Fact A or Stylized Fact X. Example: Multiple empirical observations of metabolic rate scaling support a stylized fact about allometric relationships.
- **Contradicting Evidence** (contradicts/opposes): Fact C contradicts Fact D or challenges assumptions in Stylized Fact Y. Example: Observations from tropical species contradicting temperate-zone metabolic scaling patterns.
- **Reference/Citation** (references/cites): Fact or stylized fact references another as context or prior work. Example: A new observation referencing established baseline measurements.
- **Refinement** (refines): Stylized Fact Y refines or provides more specific version of Stylized Fact X. Example: Temperature-dependent energy allocation refining general energy allocation patterns.
- **Synthesis** (synthesizes): Stylized Fact aggregates multiple facts (already captured via fact_ids array, but can be explicit edge for graph queries).

- **Methodological Context** (measured_by/estimated_by): Links facts to measurement methods or estimation techniques, enabling methodology-based queries.
- **Temporal Relationships** (precedes/follows): Time-ordered relationships between observations or developmental stages.
- **Phylogenetic/Taxonomic** (taxonomically_related): Relationships based on species relatedness, enabling comparative analysis across clades.

These typed relationships enable sophisticated queries: "Find all facts supporting this energy allocation hypothesis," "Identify contradicting evidence for metabolic scaling," "Trace refinement chain from raw observations to general principles," or "Compare evidence strength across phylogenetic groups." The relationship types capture not just that entities are related, but *how* and *why* they relate, essential for scientific knowledge synthesis and validation.

Relationship metadata includes confidence scores, curator annotations, and timestamps, allowing researchers to assess evidence quality and track how scientific consensus evolves. A supporting relationship might have high confidence (0.9) based on multiple independent replications, while a contradicting relationship might have lower confidence (0.6) based on single-study evidence requiring further investigation.

Alternative approaches were considered. A purely graph-based model (everything is nodes and edges) would force unnatural fits for narrative knowledge. A purely text-based model (everything is documents) would sacrifice structured queries. A single unified representation attempts to serve all masters and serves none well. The three-tier approach accepts redundancy costs for flexibility benefits, appropriate for a beta platform where knowledge representation patterns are still emerging.

The status field on each knowledge entity (draft, pending_review, published, rejected) enables quality control workflows without complex permissions. Creator-private drafts allow experimentation. Published knowledge is globally readable. This simple lifecycle matches research workflows where ideas incubate privately before publication.

11.2 Document Ingestion Pipeline

Design Decision: Asynchronous multi-stage ingestion with AI-powered extraction

Document uploads trigger background processing pipelines that extract text, invoke AI agents for fact extraction, and store results asynchronously.

The asynchronous pipeline reflects the reality that document processing is slow (minutes, not milliseconds) and failure-prone (PDFs are messy, AI extraction has errors). Synchronous processing would block users—uploading 50 papers would lock the interface for an hour. Background processing with status polling provides responsive UX while avoiding distributed system complexity.

The multi-stage approach (upload → text extraction → AI analysis → fact storage) allows independent failure handling. Text extraction might succeed while AI analysis fails due to API quotas. The system stores partial results and enables retry of failed stages without reprocessing successful stages. Each stage logs progress to the IngestionJob document, providing users with visibility and administrators with debugging information.

AI-powered extraction via agents represents a conscious decision to prioritize automation over accuracy. Manual fact extraction is slow but accurate; AI extraction is fast but error-prone. The platform enables both: agent_access capability users can run bulk extraction for initial knowledge population, while curator workflow allows manual fact creation and editing. The review system catches AI errors before publication, providing quality gates without sacrificing automation benefits.

The IngestionBatch abstraction enables bulk operations common in research: "upload all papers from this conference" or "ingest literature review corpus." Batches group related documents, enable batch-level status tracking, and support Day Zero knowledge seeding (administrator-designated foundational content marked is_day_zero=true). This batch model matches research workflows better than purely document-centric models.

File storage decisions involve trade-offs. Storing document content in MongoDB (content field in Document collection) simplifies backup and deployment but limits document size and wastes database storage. Storing files in object storage (S3, MinIO) improves scalability but adds operational complexity and distributed failure modes. For beta scale (thousands of documents, gigabytes total), MongoDB storage is simpler. Migration to object storage becomes necessary around tens of thousands of documents.

11.3 Agent Framework Architecture

Design Decision: LangChain-inspired agent framework with RAG and local LLM integration via Ollama

AI capabilities are provided through an agent framework that orchestrates LLM calls, Retrieval-Augmented Generation (RAG), tool invocations, and structured output parsing.

The agent framework abstracts LLM interactions behind a consistent interface: accept natural language input, retrieve relevant knowledge context, invoke necessary tools, return structured output. This abstraction enables multiple agent types (knowledge extraction, query assistance, model suggestions) to share infrastructure while specializing behavior. The AgentFramework class provides orchestration, the LocalModelClient handles Ollama communication, and the ToolRegistry enables dynamic capability extension.

Retrieval-Augmented Generation (RAG): The framework implements RAG methodology to ground LLM responses in platform knowledge. When processing queries, the system retrieves relevant facts, stylized facts, and knowledge graph entities from the knowledge base and includes them in the LLM context. This approach addresses LLM hallucination by providing factual grounding—the model generates responses based on actual curated knowledge rather than parametric memory. For document ingestion, RAG enables agents to reference existing knowledge while extracting new facts, ensuring consistency and detecting potential contradictions. For knowledge queries, RAG allows natural language questions ("What is the metabolic scaling relationship for fish?") to be answered with specific platform content rather than generic LLM knowledge.

The RAG pipeline consists of: (1) Query analysis—extract key concepts and entities from user input, (2) Knowledge retrieval—search knowledge base using text search and future vector similarity for relevant content, (3) Context assembly—format retrieved facts and stylized facts as LLM context, (4) Response generation—LLM generates response grounded in retrieved knowledge, (5) Source citation—responses include references to

specific knowledge base items for traceability. This architecture ensures scientific accountability—every AI-generated statement can be traced to curated sources.

Local LLM integration via Ollama reflects cost and privacy considerations. Cloud LLM APIs (OpenAI, Anthropic) offer better quality but incur per-token costs that scale with usage—problematic for exploratory research where token budgets are unpredictable. Ollama runs models locally (Llama 2, Mistral, etc.) with zero marginal cost once hardware is provisioned. For beta scale, a single GPU server running Ollama suffices. Cloud APIs can augment local models for high-value use cases (final publication-quality extraction) while local models handle exploratory work.

The tool system enables agents to take actions beyond text generation: search existing knowledge, query databases, call external APIs. Each tool is a Python function registered in ToolRegistry with JSON schema describing parameters. The LLM generates tool calls as structured JSON, the framework executes tools, and results feed back to the LLM for next steps. This tool-augmented generation pattern matches the research assistant metaphor: agents that can "look up information" and "perform analyses," not just generate text.

Session management tracks conversation history in AgentSession documents. Multi-turn conversations enable iterative refinement: "extract facts from this paper" → "focus on methodology section" → "create stylized fact summarizing results." Sessions store message history, tool call traces, and context (referenced fact_ids, document_ids). This history enables debugging (why did agent produce this output?) and learning (analyze successful vs. failed extraction sessions).

Alternative architectures include cloud-only (simple but expensive), model fine-tuning (accurate but slow to iterate), and no AI (accurate but manual). The local LLM + agent framework balances automation benefits, cost control, and iteration speed appropriate for research platform development.

11.4 Knowledge Query and Search

Design Decision: MongoDB text search with optional vector similarity

Knowledge search uses MongoDB's built-in text indexing for free-text queries, with architecture accommodating future vector embedding search.

MongoDB text search provides basic but fast full-text queries across fact content, document text, and graph node labels. The text index on facts.content, stylized_facts.summary, and knowledge_graphs.nodes fields enables queries like "find facts about salmon lice" with relevance scoring and result ranking. This approach leverages existing MongoDB infrastructure without additional services.

The limitation is semantic understanding—text search matches keywords but misses conceptual similarity. "sea lice" and "parasitic copepods" don't match despite referring to related concepts. Vector embeddings (encode text to numerical vectors, search by vector similarity) capture semantic relationships but require embedding models, vector storage, and similarity computation infrastructure.

The architecture accommodates future vector search by designing current text search behind an abstraction layer. The /api/knowledge/search endpoint returns scored results regardless of underlying implementation. When vector search is needed, we add embedding generation (call embedding model on knowledge insertion), store vectors (add embed-

ding field to documents), and implement similarity search (cosine similarity against query embedding). The API contract doesn't change, enabling non-breaking enhancement.

The "optional" nature reflects beta priorities: text search serves initial use cases adequately, and adding vector search complexity before proving necessity risks premature optimization. User feedback will reveal whether semantic search is essential or nice-to-have. Building the simpler solution first enables faster iteration while architecture permits enhancement.

Tag-based filtering (facts with tags=["salmon", "parasite"]) complements text search by enabling precise subset queries. Tags are manually assigned by curators or automatically extracted by agents, providing structured metadata alongside unstructured text. The combination of text search (find relevant content) and tag filtering (narrow to specific domains) matches research workflows where domain expertise guides exploration.

12 Modeling Assistant Architecture

12.1 Chatbot-Driven Model Development with RAG

Design Decision: Conversational interface with LLM and RAG for exploratory modeling

Modeling Assistant provides a chatbot interface where researchers interact with an LLM that uses RAG to traverse the knowledge base while building bioenergetic models.

The Modeling Assistant architecture centers on human-AI collaboration for developing new paradigm models in bioenergetics. Rather than requiring researchers to manually browse the knowledge base and assemble models, the chatbot interface enables natural language exploration: "What do we know about energy allocation in salmon during reproduction?" or "Show me metabolic scaling patterns across teleost fish." The LLM, augmented with RAG, retrieves relevant facts and stylized facts from the knowledge base, synthesizes information across sources, and suggests modeling approaches based on available evidence.

Conversational Model Building: The workflow reflects iterative scientific thinking. A researcher might ask exploratory questions about energy allocation patterns, request comparisons across species or life stages, examine supporting and contradicting evidence, propose model structures, and refine assumptions based on knowledge base content. The chatbot maintains conversation context across multiple turns, allowing progressive refinement. Each interaction leverages RAG to ground responses in curated knowledge—preventing hallucinations and ensuring scientific rigor.

Knowledge Base Traversal: RAG enables intelligent navigation through interconnected knowledge. When discussing energy allocation, the system retrieves related facts about metabolic rates, reproduction parameters, and transport network dynamics. It follows typed relationships (supporting evidence, contradicting evidence, refinement chains) to present comprehensive views. The chatbot can explain "This stylized fact about metabolic scaling is supported by 15 empirical observations across 8 species" while providing direct links to underlying facts. This traversal capability helps researchers discover connections and patterns they might miss through manual exploration.

Paradigm Model Development: The platform's primary focus during initial de-

velopment is creating basic tools and utilities to build a comprehensive knowledge base for bioenergetic modeling of organisms. The chatbot-RAG architecture supports this mission by making knowledge accessible and actionable. Rather than researchers spending weeks manually reviewing literature, the system surfaces relevant knowledge through conversation, enabling faster hypothesis formation and model conceptualization. The emphasis is on knowledge base construction and tooling—detailed API specifications and model execution capabilities are under active development and will evolve based on research needs.

Current Development Status: The architecture accommodates future capabilities (model simulation, parameter estimation, results visualization) while prioritizing immediate needs: knowledge ingestion, curation workflows, fact relationship management, and exploratory knowledge access through conversational interface. API endpoints for model execution and advanced analytics remain in development, with designs flexible enough to adapt as research requirements become clearer through actual usage.

The chatbot-RAG approach aligns with the platform's goal of advancing transport network-based DEB models (tDEB). By enabling researchers to explore the AmP database's 4,500 species through natural language, identify applicability domains, and discover generalities across phylogenetic groups, the system accelerates the scientific discovery process. The conversational interface lowers barriers to knowledge access, making the platform's value—comprehensive bioenergetics knowledge—immediately useful for model development.

12.2 Scenario-Based Modeling Workflow

Design Decision: Scenario entities as modeling project containers

Modeling work is organized into Scenario documents that bundle objectives, knowledge references, proposed models, and results.

The scenario-centric architecture reflects how modelers actually work. A research project doesn't start with "build a model"—it starts with "understand phenomenon X under conditions Y." Scenarios capture this: name ("Salmon lice dynamics in Norwegian fjords"), description (research context), objectives (specific questions to answer), and knowledge_queries (relevant KB searches). A scenario is a modeling project container.

This container pattern provides several benefits. Multiple models can be proposed for a single scenario, enabling comparison: "model A uses ODE formulation, model B uses IBM approach—which better matches empirical patterns?" Results link back to scenarios and models, maintaining provenance. Scenarios become the unit of collaboration: researchers share scenarios, not raw model code. The scenario UUID appears in audit logs, enabling analysis of which research questions consume most platform resources.

Alternative organizational patterns include model-centric (models are primary, scenarios implicit) or flat (no grouping). Model-centric approaches force awkward fits when exploring multiple modeling strategies for one question. Flat approaches lose project-level context that researchers need to interpret results. The scenario container matches research mental models while providing technical structure.

The knowledge_queries field links scenarios to KB content. Rather than copying knowledge into MA (duplication risk), scenarios store KB search queries that execute when scenario loads. This indirection keeps MA's knowledge view current as KB evolves, prevents

data staleness, and maintains single source of truth. The trade-off is dependency: if KB is unavailable, MA cannot resolve knowledge references. For beta scale with co-located services, this coupling is acceptable.

12.3 Model Assembly and Representation

Design Decision: Declarative model specifications with justification links to KB

Models are represented as JSON specifications describing entities, state variables, processes, and parameters, with each element justified by KB references.

The declarative specification approach separates model structure (what entities exist, what processes occur) from model execution (simulation code). A model specification document contains entities (["host", "parasite"]), state_variables (["infection_status", "age", "location"]), processes (["transmission", "recovery", "movement"]), and parameters (["beta: transmission rate", "gamma: recovery rate"]). This structured representation enables programmatic analysis, visualization, and eventually code generation.

Justification links connect model elements to KB items. The "transmission" process references KB fact IDs describing observed transmission mechanisms. The "beta" parameter references stylized facts about measured transmission rates. These links serve multiple purposes: they make modeling assumptions explicit and traceable, they enable automated literature review (what evidence supports this model?), and they facilitate model comparison (models A and B differ in transmission assumptions, which KB references support each?).

This architecture reflects scientific accountability requirements. Models without clear assumptions are black boxes; assumptions without evidence are speculation. The KB reference links transform models from opaque code to transparent reasoning chains. Reviewers can critique assumptions by examining backing evidence. This traceability overhead is justified for scientific modeling where model credibility depends on explicit reasoning.

The specification is execution-agnostic: it describes what the model represents, not how to simulate it. Future execution systems might generate NetLogo code, Python simulation loops, or mathematical equations from the same specification. This separation enables evolution: execution strategies improve without changing model semantics. For beta, specifications serve documentation and communication; execution comes later.

Alternative approaches include code-as-model (model is Python simulation code) or graphical-only (model is visual diagram). Code-as-model is precise but opaque—understanding requires reading implementation. Graphical-only is intuitive but informal—visual boxes lack semantic precision. The declarative specification with KB justification provides precision, traceability, and semantics appropriate for scientific discourse.

12.4 Integration with Knowledge Builder

Design Decision: HTTP API integration with JWT token pass-through

MA integrates with KB through HTTP APIs, passing user JWT tokens to maintain unified authentication and attribution.

The HTTP integration keeps MA and KB loosely coupled. MA makes standard HTTP GET/POST requests to KB endpoints (/api/knowledge/search, /api/knowledge/facts/{id}, /api/agents/run). This integration style enables independent deployment: KB can be updated without MA redeployment if API contracts are maintained. Development can proceed in parallel: KB and MA teams work independently with integration contracts as coordination points.

JWT token pass-through maintains security and accountability across component boundaries. When user authenticates to MA, they receive JWT token. When MA calls KB on user's behalf, it includes that token in HTTP headers. KB validates token, checks permissions, and attributes actions to original user. This preserves audit trail: knowledge queries triggered by MA scenario creation appear in audit logs attributed to the user who created the scenario, not to a generic MA service account.

The alternative of service-to-service authentication (MA has its own credentials to call KB) would simplify implementation but lose user attribution. KB would see all MA requests as coming from "MA service" rather than individual users. This breaks accountability and prevents capability-based authorization (MA can't request agent runs if user lacks agent_access capability). Token pass-through preserves end-to-end user context at cost of handling token expiration and refresh in MA-to-KB calls.

The HTTP integration introduces latency compared to direct database access (MA reading KB MongoDB directly). Each KB API call adds network round-trip and HTTP parsing overhead. For beta scale (users making requests at human speed, not high-frequency trading), this latency is negligible (tens of milliseconds). The architectural cleanliness from component separation outweighs performance costs.

Future optimizations might include caching frequently accessed KB content in MA, batching multiple KB queries, or GraphQL for flexible data fetching. These enhancements are possible because HTTP integration is standard and well-understood. Premature optimization with custom RPC or shared database access would add complexity without measurable benefit at current scale.

13 Integration Architecture and Data Contracts

13.1 Knowledge Sharing Model: API Contracts vs. Shared Database

Design Decision: MA accesses KB knowledge via HTTP APIs, not direct database access

Modeling Assistant retrieves knowledge through documented HTTP endpoints rather than reading Knowledge Builder's MongoDB collections directly.

The API-based integration enforces interface contracts between components. KB publishes documented endpoints (/api/knowledge/search, /api/knowledge/facts/{id}) with specified request/response formats. MA depends on these contracts, not internal KB data structures. This abstraction enables KB to refactor database schemas, optimize queries, or even migrate to different storage technologies without affecting MA—as long as API contracts are preserved.

Direct database sharing (both components reading/writing same MongoDB collections) would eliminate network latency and simplify queries spanning knowledge and

modeling data. However, it creates tight coupling that constrains evolution. KB cannot change fact schema without coordinating with MA. Database migrations must be synchronized across components. Concurrent writes risk data corruption without complex locking. These coupling costs would slow development velocity.

The API contract approach accepts latency costs for evolution freedom. Each KB call from MA incurs HTTP overhead (milliseconds), but development teams work independently. KB can optimize database queries without consulting MA. MA can cache KB responses locally. API versioning (v1, v2 endpoints) enables gradual migration when contracts evolve. These benefits outweigh latency costs at beta scale where users make requests at human speed.

The documented integration contracts serve as coordination points. Both teams agree on endpoint shapes, field meanings, and error responses. These explicit contracts prevent implicit dependencies and make integration points visible. Contracts are version-controlled and reviewed like code, ensuring changes are deliberate and communicated.

Alternative integration patterns include message queues (event-driven), GraphQL (flexible queries), or shared libraries (code reuse). Message queues add complexity for asynchronous workflows we don't yet need. GraphQL enables efficient data fetching but requires maintaining GraphQL server infrastructure. Shared libraries couple at code level rather than data level. Standard HTTP with JSON is simple, well-understood, and sufficient for beta needs.

13.2 Data Synchronization and Consistency

Design Decision: No data synchronization—MA stores references, not copies

MA stores KB entity IDs (fact_ids, document_ids) in scenario and model documents rather than copying knowledge content.

The reference-based approach maintains single source of truth for knowledge. When scenario references fact ID f123, MA stores "f123" not a copy of fact content. When displaying scenario, MA calls KB API to fetch current fact content. This ensures MA always shows latest knowledge version—if curators update fact f123, scenarios referencing it immediately reflect changes without synchronization logic.

The alternative of copying knowledge into MA (denormalization) would avoid KB API calls during MA operations. Scenario display would query MA's local database only, eliminating network latency and KB dependency. However, denormalization creates staleness problems: fact f123 copied into 50 scenarios requires updating 50 documents when fact changes. Synchronization logic becomes complex, error-prone, and adds operational burden.

The reference approach accepts runtime dependencies for data integrity. MA depends on KB availability—if KB is down, MA cannot resolve knowledge references and displays IDs instead of content. For beta with co-located services and high uptime expectations (99.9%), this coupling is manageable. The simplicity of avoiding synchronization outweighs availability concerns.

Caching can mitigate dependency costs without full denormalization. MA might cache KB responses (fact content) with short TTLs (5 minutes). Cache hits avoid KB calls (performance), cache expiration ensures eventual freshness (correctness). This hybrid

approach provides practical performance while maintaining reference-based semantics. Caching is optimization added when metrics show KB API latency is problematic, not premature complexity.

The cross-component transaction problem is avoided by design: MA doesn't write to KB. Scenarios and models are MA entities; facts and documents are KB entities. One-way data flow (KB produces knowledge, MA consumes knowledge) eliminates coordination for distributed transactions. If future requirements demand MA creating knowledge (e.g., "save scenario insights as KB facts"), those operations use KB's write APIs with user authentication, preserving component boundaries.

13.3 Version Compatibility and API Evolution

Design Decision: API versioning through URL path with backward compatibility expectations

KB endpoints include version prefix (e.g., /api/v1/knowledge/search) with commitment to maintain v1 contracts while developing v2.

API versioning enables contract evolution without breaking existing MA deployments. When KB needs incompatible changes (response field renamed, different semantics), it publishes v2 endpoints alongside v1. MA continues using v1 while gradually migrating to v2. Deprecated versions remain available during transition period (6 months warning before removal), providing upgrade time without forcing synchronization.

The URL path versioning (v1, v2 in path) makes versions explicit and discoverable. Alternatives like header-based versioning (Accept: application/vnd.kb.v1+json) are less visible. Query parameter versioning (?version=1) mixes versioning with query semantics. URL path versioning follows REST conventions and requires no special client configuration.

Backward compatibility within versions follows semantic versioning principles: adding optional fields is safe, changing field types is breaking, removing fields is breaking. KB can add response fields to v1 endpoints (MA ignores unknown fields) but cannot remove required fields without v2. This discipline prevents accidental breakage while allowing organic API growth.

The commitment to backward compatibility shapes development: breaking changes must justify creating new API version with all associated costs (documentation, testing, migration support). This friction encourages thoughtful API design—get contracts right initially because changes are expensive. For beta, acceptance of some API instability (rapid v1 $\rightarrow$ v2 transition) balances design flexibility with stability needs.

Documentation includes endpoint examples, response schemas (JSON schema), and error codes. This documentation is tested (automated checks that examples actually work) to prevent docs-code drift. Clear documentation enables MA development without constant KB code inspection, reducing cross-team coordination overhead.

14 Data Model and Storage Decisions

14.1 MongoDB for All Data: Knowledge Graph Foundation

Design Decision: Use MongoDB as unified storage for knowledge graphs, facts, documents, scenarios, models, and user data

The platform uses MongoDB for all persistence, primarily driven by networked knowledge graph requirements, with other data types leveraging the same infrastructure.

The MongoDB choice is fundamentally about knowledge graph scalability. The platform's value proposition is managing interconnected bioenergetics knowledge: species exhibit energy allocation patterns, organisms share metabolic scaling relationships, physiological processes interact across life stages. These relationships form a networked knowledge graph that must support typed relationship queries ("find all facts supporting this energy allocation hypothesis," "identify contradicting evidence for metabolic scaling"), graph traversal with relationship filtering ("what bioenergetic parameters are documented for salmon with high-confidence supporting evidence?"), path finding through specific relationship types ("trace the refinement chain from raw observations to general principles"), and subgraph extraction ("export knowledge subgraph for energy allocation during reproduction including all supporting and contradicting evidence").

MongoDB's document model with embedded arrays and references naturally represents graph structures. A knowledge graph node document contains {id, label, type, properties, outgoing_edges: [{target_id, edge_type, weight}]}. The edge_type field enables typed relationships essential for scientific knowledge representation: "supports" (evidence supporting a claim), "contradicts" (conflicting evidence), "refines" (more specific version), "synthesizes" (aggregates multiple sources), "references" (citation context), or domain-specific relationships like "measured_by" (methodological links) and "taxonomically_related" (phylogenetic connections). Graph traversal becomes recursive document lookups with array filtering by edge type. Aggregation pipelines implement graph algorithms: $graphLookup for breadth-first search with edge type constraints, $lookup chains for multi-hop paths following specific relationship patterns ("find all facts supporting this hypothesis through the supporting-evidence chain"). These operations leverage MongoDB's optimized B-tree indexes and in-memory aggregation pipeline.

The scalability story differentiates MongoDB from alternatives. As knowledge graphs grow (target: 100K+ entities, 1M+ relationships), MongoDB's sharding distributes graph partitions across servers. Frequently accessed subgraphs cache in memory while cold data stays on disk. Query performance remains acceptable because graph queries typically explore local neighborhoods (few hops) rather than global structures. PostgreSQL with recursive CTEs can represent graphs but doesn't scale horizontally as naturally. Neo4j excels at graph queries but requires maintaining two databases (Neo4j for graphs, something else for documents/users).

Knowledge entities (facts, stylized facts) benefit from MongoDB's flexible schema for content representation. A fact about salmon energy allocation has different fields than a fact about metabolic scaling across fish species. Stylized facts synthesize varying numbers of source facts. Document ingestion creates varying metadata depending on source type (PDF, HTML, structured data). These heterogeneous structures map to MongoDB documents naturally—each entity type has core required fields plus optional domain-specific

fields.

The JSON-native storage provides seamless integration across the stack: LLM agents output JSON knowledge structures, FastAPI receives JSON from frontends, MongoDB stores BSON (binary JSON), and HTTP APIs return JSON. This end-to-end JSON flow eliminates impedance mismatches that plague ORM-based systems where objects map awkwardly to relational rows.

User management and authentication data live in MongoDB as a secondary consideration—the database infrastructure exists for knowledge graphs, so user data uses the same infrastructure for operational simplicity. User documents are structurally simple compared to knowledge entities, making MongoDB arguably over-powered for authentication. However, introducing PostgreSQL purely for user management would add operational complexity (two databases to backup, monitor, scale) without addressing the platform's core scaling challenge: the knowledge graph.

The trade-offs are real. Complex analytical queries joining users, contributions, reviews, and knowledge entities are awkward in MongoDB, requiring multiple queries or complex aggregation pipelines. PostgreSQL would handle "find all facts created by curators with analytics access whose stylized facts were reviewed by reviewers from Institution X" with straightforward JOINs. However, such queries are rare—the platform optimizes for knowledge access patterns (text search, graph traversal, entity retrieval) not administrative analytics.

Time-series data (agent message sequences, audit logs, usage metrics) could benefit from specialized stores (TimescaleDB, InfluxDB). For beta volumes (thousands of messages per day, 10-100 active users), MongoDB's time-based indexing and TTL collections suffice. If telemetry analysis becomes central, time-series databases can augment MongoDB without replacing it—the knowledge graph requires MongoDB regardless.

The single-technology decision provides operational focus. A small research team achieves MongoDB expertise (indexing strategies, aggregation pipeline optimization, sharding configuration) rather than surface-level familiarity with multiple databases. Backup, monitoring, query optimization, and troubleshooting knowledge developed for knowledge graph storage applies everywhere. This operational efficiency matters more at beta scale than marginal performance gains from specialized databases.

Future specialization remains possible: if graph queries dominate, Neo4j could supplement MongoDB; if full-text search becomes inadequate, Elasticsearch could augment it; if analytical queries proliferate, PostgreSQL could handle reporting. But these additions come later, justified by evidence from actual usage patterns. Starting with MongoDB everywhere enables learning what the platform actually needs rather than prematurely optimizing for imagined requirements.

14.2 Document Collections and Relationships

Design Decision: Separate collections for each entity type with ID-based references

Knowledge uses separate MongoDB collections (facts, stylized_facts, documents, knowledge_graphs) with ObjectId references rather than embedded documents or single collection with type discriminators.

Separate collections reflect natural entity boundaries: facts are distinct from documents conceptually and in access patterns. Facts are queried frequently (text search, tag

filtering), documents less often (source reference lookup). Separate collections enable independent indexing: text index on facts.content, compound index on documents.uploaded_by + created_at. Single collection with type field would require sparse indexes and complex queries filtering by type.

ID-based references (fact.document_id references documents._id) mirror relational foreign keys but without enforcement. MongoDB doesn't enforce referential integrity—deleting document doesn't cascade delete its facts. This lack of enforcement is pragmatic: research workflows might intentionally preserve facts after document removal (document found corrupt but facts are valid). Explicit application-level handling of deletions provides flexibility at cost of manual integrity maintenance.

The alternative of embedding (documents contain embedded facts array) would avoid reference lookups but doesn't fit access patterns. Facts are accessed independently via search, not always through parent document. Embedding facts in documents would require querying documents collection with array element matching, inefficient for fact-centric queries. Embedding only makes sense for truly nested data accessed together (agent messages embedded in agent sessions).

Relationship modeling follows MongoDB best practices: one-to-many with references (document has many facts: facts.document_id), many-to-many with arrays (knowledge graph contains many stylized facts: knowledge_graphs.stylized_fact_ids = [id1, id2, ...]). This hybrid approach (references for one-to-many, arrays for many-to-many) balances query efficiency and update convenience.

The User relationship pattern (created_by: ObjectId referencing users collection) links knowledge to creators for attribution. Every knowledge entity has created_by field added through authentication integration, enabling queries like "facts created by curator X" for contribution tracking and quality analysis. This universal attribution pattern enables future features (reputation systems, recommendation based on contributor expertise) without schema changes.

15 Future Evolution

15.1 What Initial Implementation Deliberately Excludes

Design Decision: Conscious omissions from initial scope

Several features commonly found in authentication systems are deliberately excluded from the initial implementation, not forgotten or overlooked.

Multi-factor authentication beyond what Google provides is not implemented because it's redundant. Users who enable 2FA on their Google accounts bring that protection to AdvanDEB automatically through OAuth. Implementing separate TOTP (Time-based One-Time Password) authentication would add complexity and require users to manage another device/app configuration. If future requirements demand platform-specific MFA (perhaps for high-privilege operations like approving financial transactions), the authentication architecture accommodates adding it without fundamental redesign.

IP whitelisting for API keys is not implemented because most users lack static IPs. Academic institutions often use dynamic addressing. Researchers work from home, coffee shops, conferences—their IP addresses change constantly. Implementing IP whitelisting would create support burden (legitimate users locked out) without meaningful security

benefit (sophisticated attackers use proxies anyway). The rare use cases requiring IP restriction (API keys for server-to-server integration) can be handled through manual configuration if they arise.

Fine-grained ACLs (access control lists) on individual knowledge items are not implemented because current requirements don't justify the complexity. All published knowledge is public—any curator can read it. Draft content is private to its creator until submitted for review. This binary model (public or creator-private) handles current needs without the overhead of per-document permission checking. If future requirements add team workspaces or private knowledge collections, the capability model extends naturally—a "workspace_member" capability could grant access to workspace documents.

Role hierarchies with inheritance are not implemented because they add complexity that capability composition avoids. In hierarchical models, "senior curator" might inherit from "curator" plus additional permissions. This inheritance creates coupling—changing base role permissions affects all derived roles. The capability model achieves similar results through composition (curator + analytics + reviewer) without inheritance complexity. Capabilities combine independently without creating dependency graphs that become difficult to reason about.

These omissions reflect YAGNI (You Aren't Gonna Need It) principle from agile development. Building features before they're needed often means building the wrong features because requirements understanding evolves through use. The authentication architecture is extensible—none of these omissions painted us into corners. If evidence emerges that IP whitelisting or fine-grained ACLs are needed, they can be added without fundamental redesign. Until that evidence appears, simpler is better.

16 Conclusion

16.1 Comprehensive Platform Architecture

This document now provides comprehensive architectural rationale for the AdvanDEB Platform across all major components:

- **User Management & Authentication**: Google OAuth integration, JWT tokens, capability-based roles, API keys, audit logging, and cross-component SSO
- **Knowledge Builder**: Semi-supervised knowledge base construction through chatbot-mediated document processing and knowledge extraction, three-tier knowledge representation (Facts, Stylized Facts, Graphs), asynchronous document ingestion pipeline, AI agent framework with RAG and local LLM integration, and MongoDB text search
- **Modeling Assistant**: Chatbot interface with LLM and RAG for conversational knowledge exploration and model development, scenario-based workflow for organizing modeling projects, declarative model specifications with KB justification links
- **Integration Architecture**: API-based component coupling, reference-based data model avoiding synchronization, and versioned HTTP contracts
- **Data Model**: MongoDB for all data types, separate collections per entity, ID-based references, and universal creator attribution

The architectural decisions across these components follow consistent principles established in the authentication layer: simplicity over premature optimization, proven technologies over novel approaches, loose coupling with explicit contracts, and evolvability through abstraction layers. Each component can evolve independently while maintaining platform coherence through shared authentication, consistent data attribution, and documented integration contracts.

16.2 Architectural Principles Synthesized

The authentication architecture emerges from applying consistent principles across all design decisions. Simplicity was prioritized over cleverness—proven technologies like MongoDB and JWT tokens over novel but unproven approaches. Security by default rather than security by configuration means users can't accidentally weaken protections through misconfiguration. User empowerment through the capability model allows organic permission evolution rather than rigid role boundaries. Evolvability ensures that future requirements can be addressed through extension rather than redesign.

These principles sometimes conflict, requiring explicit trade-offs. The shared user database sacrifices some component independence for consistency and simplicity. The capability model sacrifices permission granularity (no per-document ACLs) for approval workflow simplicity. JWT tokens sacrifice immediate revocation capability for stateless performance. Each trade-off was made consciously after examining alternatives and understanding implications.

The architecture is not optimal by any universal metric—no architecture is. It's optimal for AdvanDEB's specific context: a scientific platform requiring accountability, serving academic users with modest technical sophistication, built by a small team prioritizing feature development over infrastructure complexity, scaling to 10-20 users initially in beta with capacity for approximately 100 users. Different contexts would justify different decisions.

16.3 Success Criteria for Authentication Architecture

The authentication architecture succeeds if users authenticate easily and work across components seamlessly, administrators manage permissions efficiently without drowning in approval requests, developers add features without breaking authentication, and the platform scales smoothly from 10-20 beta users to approximately 100 users while maintaining 99.9% uptime and sub-100ms authentication latency. Security success means zero authentication-related breaches—no stolen passwords (we don't have them), no privilege escalation exploits, no audit log gaps that hide malicious behavior.

These criteria are measurable and time-bound. User experience quality shows in support ticket volume (fewer tickets about authentication problems indicates better design). Administrator efficiency shows in time spent on approval workflows. Developer productivity shows in feature velocity and bug rates. System performance shows in monitoring dashboards. Security success shows in audit results and incident records (or lack thereof).

The architecture is a means to these ends, not an end itself. Beautiful architecture that frustrates users or slows development fails regardless of technical elegance. Pragmatic architecture that achieves project goals succeeds even if it wouldn't win design awards. This document exists to explain how architectural decisions connect to project success, making the pragmatism behind each choice explicit and justifiable.

16.4 Architecture as Living Design

This is not the final architecture. The decisions documented here represent our best current understanding based on requirements analysis, user research, and technical evaluation. However, architecture must evolve as implementation reveals unexpected challenges, user feedback exposes usability issues, or performance testing uncovers bottlenecks.

Real-world experience often contradicts theoretical assumptions. A database query pattern that seemed efficient during design may prove problematic under actual usage patterns. An authentication flow that seemed intuitive on paper may confuse real users. An integration approach that appeared clean architecturally may create operational friction during deployment.

The AdvanDEB architecture is designed for evolution. Component boundaries, API contracts, and abstraction layers provide flexibility to revise implementations without complete rewrites. MongoDB's schema flexibility accommodates data model adjustments. The capability model supports permission changes without code modifications. JWT's stateless design allows authentication infrastructure changes without disrupting active sessions.

We expect this document to evolve alongside the platform. Significant architectural changes will trigger document updates with version history tracking design evolution. Failed experiments will be documented alongside successes to preserve institutional knowledge about what didn't work and why. The goal is not architectural perfection from the start, but architectural learning throughout development.

16.5 Future Architectural Enhancements

While this document covers the core platform architecture, several areas will require additional architectural decisions as implementation proceeds:

1. **Advanced Search Capabilities**: Vector embeddings for semantic search, hybrid search combining text and vectors, relevance tuning for scientific literature

2. **Collaboration Features** (Phases 3-4): Team workspaces, collaborative knowledge editing, shared scenario development, contribution conflict resolution

3. **Model Execution**: Simulation engine integration, code generation from model specifications, parameter estimation, results visualization

4. **Performance Optimization**: Caching strategies, database indexing tuning, query optimization, background job processing with Celery

5. **Advanced Agent Capabilities**: Multi-agent workflows, tool composition, agent result verification, incremental learning from corrections

6. **Trust and Reputation Systems**: Contributor reputation scoring, knowledge quality metrics, automated fact checking, citation network analysis

7. **Plugin Architecture**: Extension points for custom tools, agent plugins, visualization plugins, third-party integrations

These enhancements build upon architectural foundations documented here. The decision to defer these features reflects beta priorities: establish core platform functionality

(knowledge management, basic modeling support, user management) before adding advanced capabilities. The architecture accommodates these enhancements through extension points designed into current systems.

Document Version: 12.25.1
Date: December 12, 2025
Status: Complete Architecture Rationale (All Platform Components)
Target Scale: Beta with 10-20 users, maximum 100 users
Architecture Status: Living document - subject to revision based on implementation experience
Contact: AdvanDEB Development Team