1 Design Assumptions

1.1 User Roles and Characteristics Assumptions

The system design assumes the existence of two primary user roles:

- **Freelancer (Primary User):**

  The primary users are professional freelancers who actively use the Upwork platform to apply for jobs. It is assumed that users:

  - Have basic familiarity with web browsers and browser extensions.
  - Understand the standard Upwork workflow (viewing jobs, analyzing clients, submitting proposals).
  - Possess at least a minimal professional profile including skills, experience, and portfolio items.
  - Are capable of reviewing AI-generated content and making final decisions manually.

- **Administrator (Secondary/Internal Role):**

  The administrator role is assumed to be limited to internal system maintenance tasks such as monitoring system health, managing AI API keys, and reviewing global usage metrics. Administrators are assumed to have technical expertise and are not typical end-users of the system.

No assumptions are made about user technical expertise beyond basic computer literacy. The system is explicitly designed to assist decision-making rather than replace human judgment, aligning with the SRS system boundary definitions.

1.2 Deployment Environment Assumptions

The system design assumes that the **Upwork Proposal Assistant** is deployed as a **browser extension** operating in modern web browsers. The following assumptions apply:

- Users access Upwork through supported browsers such as **Google Chrome, Mozilla Firefox, Microsoft Edge, etc**.
- The extension is built according to **Manifest V3 specifications**, ensuring compatibility with current browser security models.
- The user environment includes:
  - Stable internet connectivity for AI-based proposal generation.
  - Sufficient system resources to support lightweight background processing.
- Backend services are deployed on a cloud-based infrastructure (e.g., AWS or GCP), providing scalability, availability, and geographic distribution.

The design assumes no dependency on operating-system-specific APIs, making the system OS-agnostic at the application level.

1.3 Usage Pattern Assumptions

The system is designed under the assumption that:

- Users interact with the extension **on-demand**, primarily when viewing Upwork job postings.
- Typical usage involves:
  - Viewing a job post
  - Triggering client analysis
  - Reviewing a reliability score
  - Generating and refining a proposal draft
- The system is **not used continuously** but in short, focused sessions aligned with job application workflows.
- AI-powered proposal generation is invoked selectively rather than excessively to avoid unnecessary API usage.

It is assumed that users prefer **fast feedback**, minimal UI interruptions, and the ability to manually edit generated proposals before submission.

1.4 Data Availability Assumptions

The system design assumes that:

- Required client and job data are available within the **Upwork job page DOM** and can be extracted using stable selectors.
- The structure of Upwork job pages remains reasonably consistent over time.
- External AI services (e.g., LLM APIs) are available and responsive during normal operation.
- Users provide accurate and up-to-date profile information (skills, experience, preferences) for effective proposal personalization.

The system does not assume access to private Upwork APIs or sensitive user credentials, in compliance with platform policies.

2 Design Constraints

2.1 Technology Stack Constraints

The design is constrained by the selected technology stack, which includes:

- **Browser Extension Technologies:** JavaScript, HTML, CSS
- **Extension APIs:** WebExtensions API (Manifest V3)
- **Backend:** RESTful APIs implemented using server-side technologies
- **AI Integration:** External AI service APIs

- **Database:** Cloud-hosted NoSQL database (e.g., MongoDB)

These constraints limit the system to technologies that are:

- Secure
- Widely supported
- Compatible with browser extension security policies

2.2 Platform Constraints (Browser Extension Limitations)

As a browser extension, the system is constrained by:

- **Sandboxed execution environments**
- Restricted access to system-level resources
- Limited background processing capabilities
- Strict permission declarations required by browsers
- Content script limitations when interacting with third-party websites

These constraints influenced the separation of responsibilities between content scripts, background scripts, and backend services.

2.3 Performance Constraints

Performance requirements defined in the SRS impose strict constraints, including:

- UI responsiveness within defined time limits
- Analysis and proposal generation within acceptable latency thresholds
- Limited CPU and memory usage to prevent browser degradation

These constraints guided the decision to offload computationally expensive tasks (e.g., AI processing) to backend services rather than executing them locally.

2.4 Security Constraints

Security-related constraints include:

- Mandatory encryption for data in transit (TLS)
- Secure local storage of sensitive data
- No storage of user credentials
- Compliance with data privacy principles and anonymization requirements

These constraints restrict how data is stored, processed, and transmitted, directly influencing backend API and storage design.

2.5 Business and Timeline Constraints

The system design is constrained by:

- Academic project timelines
- Limited development resources
- Requirement to strictly follow the approved SRS
- Evaluation criteria defined in Project Milestone 3

These constraints required prioritization of core functionalities and avoidance of scope creep.

3 Key Design Decisions

This section explains the rationale behind critical design choices, demonstrating traceability between requirements and design artifacts.

3.1 DFD Decomposition Strategy

3.1.1 Process Breakdown Rationale

The Data Flow Diagrams were decomposed incrementally from **Level 0 to Level 2** to manage system complexity. High-level processes were broken down based on:

- Functional responsibility
- Data transformation boundaries
- Logical separation of concerns

For example, proposal generation was decomposed into data extraction, skill matching, prompt construction, AI interaction, and result presentation.

3.1.2 Balancing Approach

Balancing was ensured by maintaining consistency of inputs and outputs across DFD levels. Each child diagram preserved the data flows of its parent process, ensuring:

- No data loss
- No introduction of undocumented processes
- Full traceability to functional requirements

3.2 Class Relationship Decisions

3.2.1 Association, Aggregation, and Composition

- **Association** was used where objects interact without ownership (e.g., User and Job).

- **Aggregation** was applied where objects logically belong together but can exist independently (e.g., User and Template).
- **Composition** was used where lifecycle dependency exists (e.g., Proposal and ProposalDraft).

These choices reflect real-world ownership semantics and reduce coupling.

3.2.2 Inheritance Decisions

Inheritance was used selectively to avoid overgeneralization. Shared attributes and behaviors were abstracted into base classes (e.g., User → Freelancer/Admin) only where reuse was meaningful.

3.2.3 Interface Design Decisions

Interfaces were introduced to abstract external services such as AI providers, allowing:

- Vendor independence
- Easier testing
- Future extensibility

3.3 Functionality Distribution

3.3.1 Multiple Sequence Diagrams

Multiple sequence diagrams were created to:

- Avoid overcrowded diagrams
- Clearly represent distinct use cases
- Improve readability and traceability

Each sequence diagram maps to a specific functional requirement.

3.3.2 Multiple Activity Diagrams

Activity diagrams were separated by workflow type (analysis, generation, template management) to:

- Clearly show decision points
- Model parallel activities
- Improve understanding of user-system interaction

3.3.3 Component Separation Rationale

The system was divided into components (UI, analysis engine, AI service, storage) to:

- Reduce coupling
- Improve maintainability

- Support scalability

3.4 Architecture Decisions

3.4.1 Browser Extension Architecture

A **three-part extension architecture** (content scripts, background scripts, UI components) was chosen to comply with Manifest V3 and ensure security.

3.4.2 Backend API Design Decisions

RESTful APIs were selected for:

- Stateless communication
- Simplicity
- Scalability
- Compatibility with browser-based clients

3.4.3 Database Design Decisions

A NoSQL database was chosen to support:

- Flexible data models
- Rapid schema evolution
- Efficient storage of semi-structured data