Unsprawl: Technical Documentation

Technical Lead: Jerome Rodrigo

Unsprawl is an AI-powered task management platform that consolidates tasks from multiple sources (Slack, Trello, Jira) and uses semantic similarity detection to identify and deduplicate similar tasks. This document provides comprehensive technical documentation for project handoff, covering architecture, database schema, authentication flows, integration pipelines, duplicate detection engine, frontend architecture, API reference, development guide, and deployment considerations.

December 01, 2025

Contents

1. Summary

Unsprawl is an AI-powered task management platform that consolidates tasks from multiple sources (Slack, Trello, Jira) and uses semantic similarity detection to identify and deduplicate similar tasks. The system helps teams and project managers stay organized by providing a unified view of all tasks across different tools.

1.1. What Unsprawl Does

Unsprawl addresses the common problem of task sprawl across multiple platforms. It:

- **Integrates** with Slack, Trello, and Jira to pull tasks from all sources
- **Extracts** tasks from Slack messages using Google Gemini LLM
- **Consolidates** tasks into a unified database with standardized schema
- **Detects duplicates** using semantic embeddings (SBERT[1]) and pgvector[2] similarity search
- **Validates duplicates** using Google Gemini as an LLM judge[3]
- **Provides** a Streamlit dashboard for visualization and task management

1.2. Tech Stack Overview

Backend:
- FastAPI (REST API)
- PostgreSQL with pgvector (For vector similarity search)
- SQLAlchemy (ORM) with Alembic (migrations)
- Sentence Transformers (SBERT) for task embeddings
- DeBERTa-v3[4] zero-shot classification model for Slack message classification
- Google Gemini API (LLM for task extraction and duplicate validation)

Frontend:
- Streamlit (Python-based web framework)
- Altair, Plotly (data visualization)
- Pandas (data manipulation)

Infrastructure:
- Docker & Docker Compose (containerization)
- Digital Ocean (deployment)

1.3. Implemented Features

The Unsprawl platform includes the following completed features and functionality:

Authentication & User Management:
- User registration and login with secure session management
- Session-based authentication with token storage
- Password hashing using bcrypt

Integration Management:

[1] See Glossary
[2] See Glossary
[3] See Glossary
[4] See Glossary

- OAuth 2.0[5] integration for Trello, Jira, and Slack
- Secure token storage and management
- Support for multiple integrations per user
- Integration metadata storage for service-specific information

Data Synchronization:
- Automated data fetching from Slack, Trello, and Jira
- Raw data storage in dedicated tables (`slack_raw` , `trello_raw` , `jira_raw`)
- Incremental sync capabilities with configurable time ranges
- Processing flags to track data transformation status
- Big Sync: One-click comprehensive sync that syncs all integrations, generates embeddings, detects duplicates, and validates with Gemini in a single operation

Task Extraction & Processing:
- Task extraction from Slack messages using Google Gemini LLM
- Direct mapping from Trello cards and Jira issues to standardized task format
- Unified task schema across all sources
- Source tracking via foreign key relationships

Embedding[6] Generation:
- SBERT-based embedding generation using `all-mpnet-base-v2` model
- 768-dimensional vector embeddings stored in PostgreSQL with pgvector
- Batch processing for efficient embedding generation
- Automatic embedding for new tasks

Duplicate Detection:
- Semantic similarity search using pgvector cosine similarity[7]
- HNSW[8] indexing for fast approximate nearest neighbor search
- Duplicate group creation with configurable similarity thresholds
- LLM-based validation using Google Gemini to confirm duplicates

Frontend Dashboard:
- Streamlit-based web interface with multiple pages
- Task metrics and analytics visualization
- Task management interface with filtering and sorting
- Duplicate management interface for reviewing and resolving duplicates
- Integration management page for connecting/disconnecting services
- Big Sync button in sidebar for comprehensive one-click synchronization

API & Backend Services:
- RESTful API with FastAPI
- Comprehensive API documentation via Scalar
- Service layer architecture for business logic
- Repository pattern for data access

---

[5] See Glossary
[6] See Glossary
[7] See Glossary
[8] See Glossary

1.4. Quick Start Guide

1. Clone the repository and navigate to the project directory
2. Set up `.env` file[9]
3. `docker compose up --build`
4. `docker compose run --rm migrate`
5. `http://localhost:8000`
6. `streamlit run frontend/app.py`
7. `http://localhost:8000`

[9] See Development Guide

2. Architecture Overview

2.1. System Architecture

Unsprawl follows a 3-tier layered architecture:

Presentation Layer (Frontend):
- Streamlit application providing user interface
- Communicates with backend via REST API
- Manages user sessions and authentication state

Application Layer (Backend):
- FastAPI REST API handling business logic
- Service layer for integrations, embeddings, and deduplication
- Repository pattern for data access

Data Layer:
- PostgreSQL database with pgvector extension
- Stores users, sessions, integrations, raw data, tasks, embeddings, and duplicate groups

2.2. System Diagram

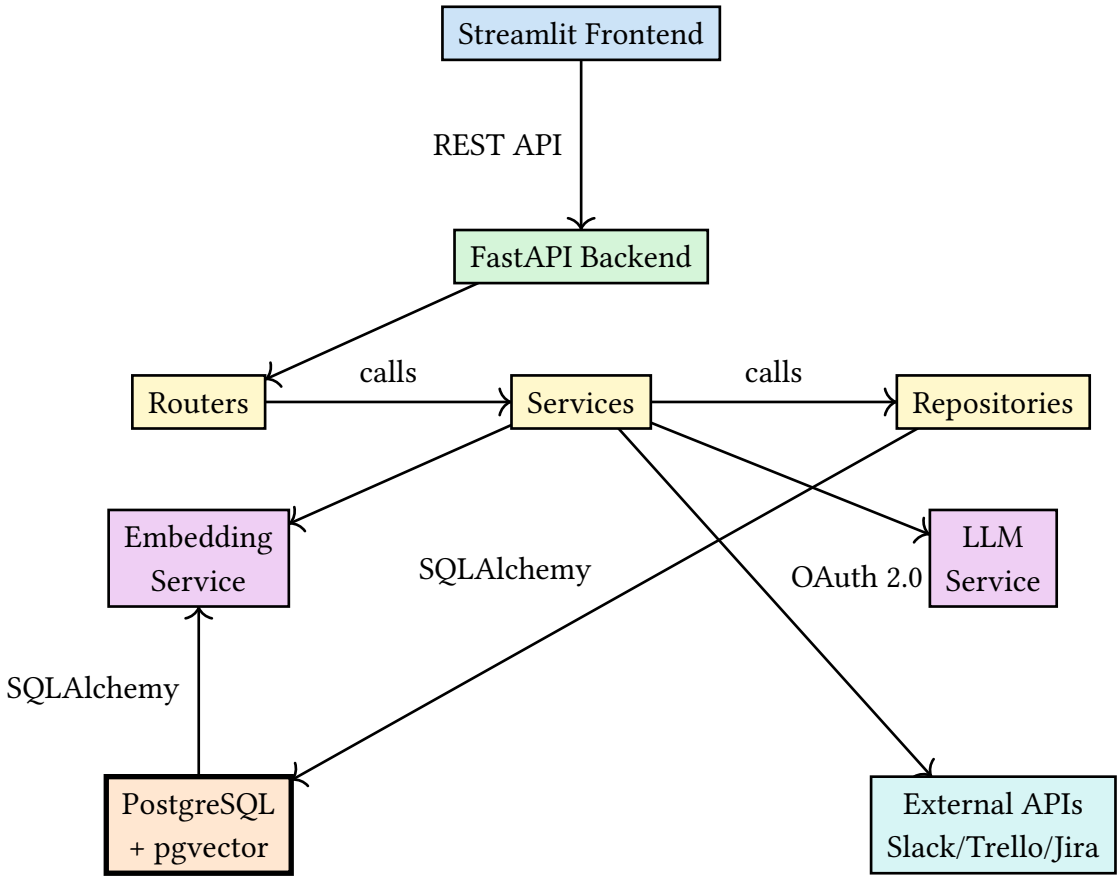

Figure 2.1: Unsprawl System Architecture

2.3. Data Flow

The system processes data through the following pipeline:

1. **Integration** → User connects Slack/Trello/Jira via OAuth

2. **Sync** → Backend fetches raw data from external APIs
3. **Raw Data Storage** → Data stored in `slack_raw`, `trello_raw`, `jira_raw` tables
4. **Task Extraction** → Raw data transformed into standardized `tasks` table
   - For Slack: Gemini extracts tasks from messages
   - For Trello/Jira: Direct mapping from cards/issues
5. **Embedding Generation** → SBERT generates 768-dim vectors, stored in `task_embeddings`
6. **Duplicate Detection** → pgvector finds similar tasks using cosine similarity
7. **LLM Validation** → Gemini validates duplicate groups
8. **Duplicate Groups** → Confirmed duplicates stored in `duplicate_groups` table

2.4. Technology Choices

FastAPI: Chosen for async/await support and automatic API documentation via Scalar. Essential for handling multiple concurrent API requests from integrations.

PostgreSQL + pgvector: PostgreSQL provides a relational database. pgvector enables efficient vector similarity search using HNSW indexing, perfect for finding duplicate tasks at scale.

SBERT (Sentence Transformers): Provides semantic understanding of task content, allowing detection of duplicates even when wording differs. The `all-mpnet-base-v2` model offers a good balance of accuracy and performance.

Google Gemini: Used for both task extraction (from Slack messages) and duplicate validation. Provides reasoning capabilities beyond simple similarity matching.

Streamlit: Python-based web framework for simple frontend development, without knowledge of frontend JavaScript and other frameworks.

SQLAlchemy: Python ORM that provides database abstraction and simplifies database operations, and works well for database queries.

Alembic: Database migration tool for SQLAlchemy.

Digital Ocean: Cloud platform chosen for deployment. App Platform provides containerized deployment with automatic scaling, integrated CI/CD, and straightforward configuration.

Supabase: Managed PostgreSQL with pgvector extension support, eliminating the need for self-hosted database management.

2.5. Key Design Decisions

2.5.1. Why pgvector for Embeddings

pgvector provides native vector operations in PostgreSQL, enabling:
- Efficient cosine similarity search using `<=>` operator
- HNSW indexing for fast approximate nearest neighbor search
- No need for separate vector database, simplifies architecture

2.5.2. Why Arrays for Duplicate Groups

Using `ARRAY(Integer)` for `task_ids` instead of a junction table allows for:
- Simpler queries (no joins needed)
- Atomic updates (entire group updated in one operation)

- GIN (Generalized Inverted Index) indexing enables O(n log n) search performance for array containment queries
- Unique constraint on `(user_id, task_ids)` ensures no duplicate groups exist

Why this trade-off is acceptable:
- Duplicate groups are typically small (2-7 tasks per group), so array size is manageable
- The performance benefits (simpler queries, atomic updates) outweigh normalization concerns for this read-heavy, group-based access pattern

GIN Indexing for Array Search: A GIN index is created on the `task_ids` array column, which allows PostgreSQL to efficiently check if a task ID exists in any group. This enables fast lookups when checking if a task is already part of a duplicate group, with O(n log n) time complexity instead of $O(n^2)$ for sequential scans.

Mutual Exclusivity Constraint: Each task can only belong to one duplicate group at a time. This is enforced by:
1. The unique constraint `(user_id, task_ids)` prevents duplicate groups from being created
2. The deduplication service checks existing groups before creating new ones
3. Task IDs are sorted before insertion to ensure consistent group representation (e.g., `[1, 5, 12]` is the same as `[12, 1, 5]`)

This ensures data integrity and prevents conflicting duplicate relationships.

2.5.3. Integration Metadata Structure

The `integration_metadata` JSON field stores the following service-specific information::
- **Jira**: `cloud_id` , `url` , `name` for each accessible site
- **Slack**: `team_id` , `team_name` for workspace identification
- **Trello**: Not used (board and list information stored in `trello_raw` table instead)

This flexible schema allows adding new integration types without schema changes.

3. Authentication & OAuth Flows

3.1. User Authentication

Unsprawl uses session-based authentication. When a user logs in:

1. User provides email and password
2. Backend verifies credentials using bcrypt password hashing
3. Backend creates a session record in `sessions` table
4. Backend returns `session_token` to frontend
5. Frontend stores token and includes it in subsequent API requests
6. Backend validates token on each request via `get_current_user` dependency

Session tokens[10] expire after a configured time period. The frontend handles token refresh automatically.

3.2. OAuth 2.0 Flows

Each integration (Trello, Jira, Slack) uses OAuth 2.0 for secure authentication without storing user passwords.

3.2.1. Trello OAuth Flow

Trello uses a simplified OAuth flow:

1. **Initiate**: User clicks "Connect Trello" → Frontend calls `GET /trello/auth/connect`
2. **Redirect**: Backend generates OAuth state and returns Trello authorization URL
3. **Authorize**: User authorizes on Trello's website
4. **Callback**: Trello redirects to `GET /trello/auth/callback` with token
5. **Complete**: Frontend calls `POST /trello/auth/complete` with token
6. **Store**: Backend stores token in `tokens` table and creates `integration` record

3.2.2. Jira OAuth Flow

Jira uses standard OAuth 2.0 with authorization code exchange:

1. **Initiate**: User clicks "Connect Jira" → Frontend calls `GET /jira/auth/connect`
2. **State Generation**: Backend creates OAuth state record for CSRF protection
3. **Redirect**: User redirected to Jira authorization URL with state parameter
4. **Authorize**: User authorizes on Jira's website
5. **Callback**: Jira redirects to `GET /jira/auth/callback` with `code` and `state`
6. **Exchange**: Backend exchanges authorization code for access token
7. **Resources**: Backend fetches accessible Jira sites using access token
8. **Store**: Backend stores token and site metadata, creates integration record

3.2.3. Slack OAuth Flow

Slack uses OAuth 2.0 v2 API:

1. **Initiate**: User clicks "Connect Slack" → Frontend calls `GET /slack/auth/connect`
2. **State Generation**: Backend creates OAuth state record

[10] See Glossary

3. **Redirect**: User redirected to Slack authorization URL
4. **Authorize**: User authorizes workspace access on Slack's website
5. **Callback**: Slack redirects to `GET /slack/auth/callback` with `code` and `state`
6. **Exchange**: Backend exchanges code for access token via `oauth.v2.access` endpoint
7. **Verify**: Backend verifies state and fetches workspace info
8. **Store**: Backend stores token and workspace metadata

3.3. Token Refresh Strategy

Jira: Supports refresh tokens. Backend can refresh expired access tokens using `refresh_token` stored in `tokens` table.

- For production, implement automatic token refresh for Jira before token expiration.

Slack: Tokens are long-lived. Refresh handled manually by user re-authenticating if needed.

Trello: Tokens don't expire. No refresh mechanism needed.

4. Integration Pipelines

4.1. Slack Integration

4.1.1. API Endpoints Used

- `conversations.list` - List all channels
- `conversations.history` - Fetch messages from channels
- `oauth.v2.access` - OAuth token exchange
- `auth.test` - Verify token validity

4.1.2. Data Transformation

Slack Message → SlackRaw:

- Direct mapping: `username`, `channel`, `text`, `timestamp`
- Thread handling: `thread_ts` and `reply_count` for threaded conversations
- `is_processed` set to `false` initially (marks whether message has been transformed into a task)

SlackRaw → Task: This transformation uses a two-stage approach:

1. **Classification Stage**: DeBERTa-v3 zero-shot classifier determines if the message contains a task
   - Model: `MoritzLaurer/deberta-v3-base-zeroshot-v2.0`
   - Labels: `["task", "casual conversation"]`
   - Threshold: Messages with task score > 0.45 are classified as tasks
   - Non-task messages are skipped to reduce unnecessary LLM calls
2. **Extraction Stage**: Google Gemini extracts task details from classified task messages
   - Prompt: "Extract actionable tasks from this Slack message…"
   - Gemini returns structured task data (title, description, **optional**: due date)
   - Creates Task record with `slack_message_id` linking to source
   - Marks `slack_raw.is_processed = true`

4.1.3. Example Transformation

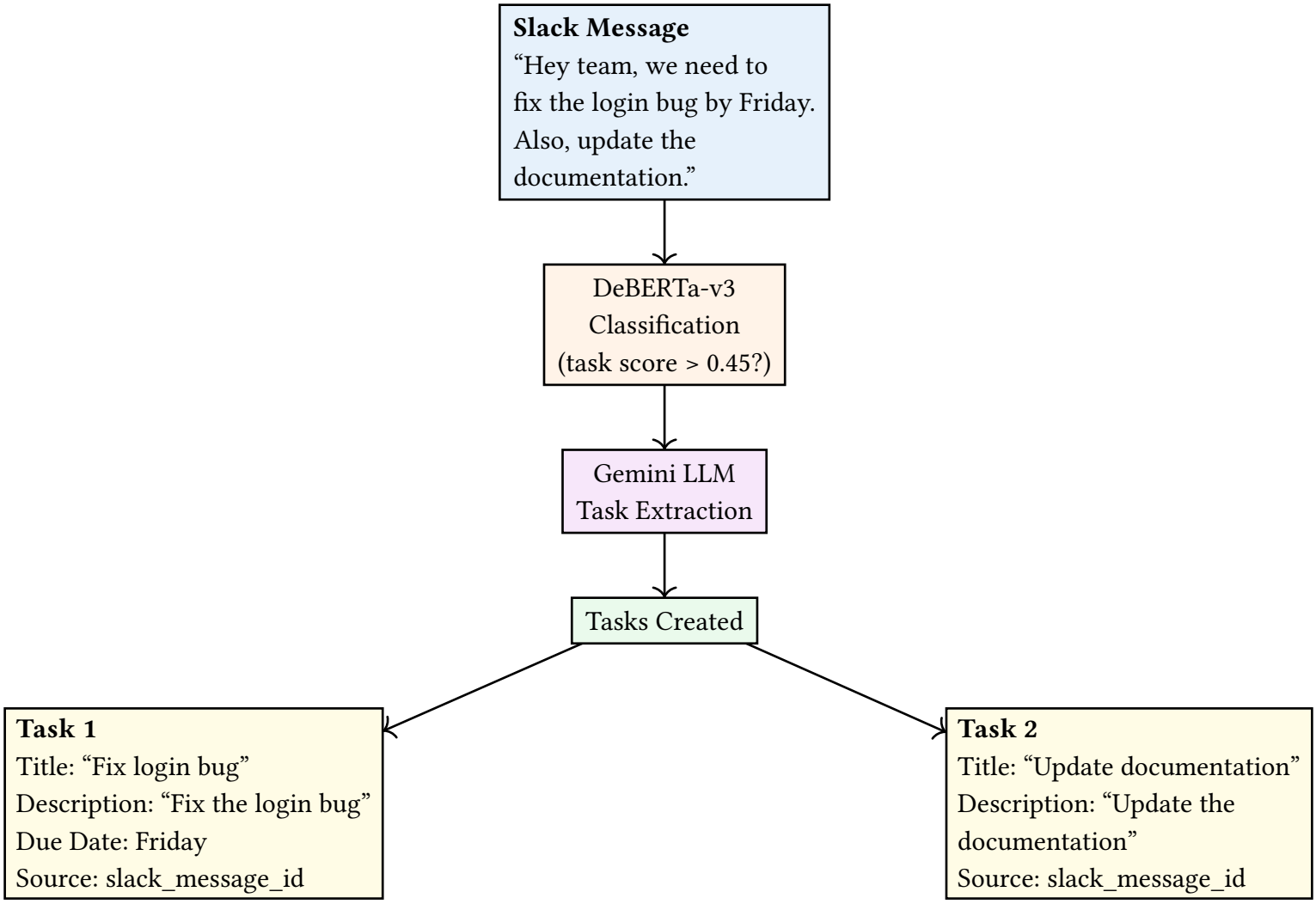

Figure 4.1: Slack Message to Task Extraction Example

4.1.4. Task Classification Model

Unsprawl uses a zero-shot classification model to filter Slack messages before task extraction. This two-stage approach significantly reduces LLM API costs by only processing messages that are likely to contain tasks.

Model Details:

- **Model**: DeBERTa-v3 Base (`MoritzLaurer/deberta-v3-base-zeroshot-v2.0`)
- **Task**: Zero-shot text classification
- **Framework**: Hugging Face Transformers pipeline
- **Device**: CPU (device=-1)

Classification Process:

1. Each Slack message is classified into two categories: `"task"` or `"casual conversation"`
   - Multi-label classification is enabled to capture messages that might be both
2. The model uses a hypothesis template: `"This message is a {}"`
3. A task score threshold of 0.45 is applied. If the score is:
   - **Greater** than 0.45: messages proceed to LLM extraction
   - **Less** than 0.45: messages are skipped, saving API costs

Why Zero-Shot Classification?

- No training data required - works out of the box
- Can classify messages without fine-tuning on Slack-specific data
- Fast inference on CPU
- Reduces false positives by filtering out casual conversations

Performance Benefits:

- Reduces LLM API calls by 60-70% (estimated based on typical Slack message distribution)
- Faster processing pipeline (classification is much faster than LLM extraction)
- Lower operational costs for high-volume Slack workspaces

Example Classification:

```
Message: "Hey team, we need to fix the login bug by Friday"
→ Task score: 0.87 → Proceeds to Gemini extraction

Message: "Thanks for the update!"
→ Task score: 0.12 → Skipped (casual conversation)
```

4.1.5. Edge Cases

- **Empty messages**: Skipped during processing
- **Thread replies**: Processed separately from parent message
- **Missing fields**: Default values used (e.g., empty description becomes "No description")
- **Classification failures**: If classifier fails to load, all messages are treated as potential tasks

4.2. Trello Integration

4.2.1. API Endpoints Used

- `GET /1/members/me/boards` - List user's boards
- `GET /1/boards/{boardId}/cards` - Fetch cards from board
- `GET /1/cards/{cardId}` - Get card details
- `GET /1/boards/{boardId}/lists` - Get board lists

4.2.2. Data Transformation

Trello Card → TrelloRaw:

- Maps Trello card fields: `card_name`, `card_desc`, `board_id`, `board_name`, `list_id`, `due`, `due_complete`, `closed`, `date_last_activity`, `card_members`
- `card_id` used as unique identifier
- `is_processed` set to `false`

TrelloRaw → Task:

- `card_name` → `task.title`
- `card_desc` → `task.description`
- `due` → `task.due_date` (parsed from string)
- `list_id` → `task.status` [11]
- `trello_card_id` stored for source tracking
- Marks `trello_raw.is_processed = true`

[11] Mapped: "To Do" → "pending", "In Progress" → "in_progress", "Done" → "completed"

4.2.3. Edge Cases

- **Archived cards**: `closed=True` cards are synced but marked appropriately
- **Missing descriptions**: Default to "No description"

4.3. Jira Integration

4.3.1. API Endpoints Used

- `GET /oauth/token/accessible-resources` - List accessible Jira sites
- `GET /rest/api/3/search` - Search issues (JQL[12] query)
- `GET /rest/api/3/issue/{issueIdOrKey}` - Get issue details
- `POST /oauth/token` - Token refresh

4.3.2. Data Transformation

Jira Issue → JiraRaw:

- Maps Jira issue fields to raw table
- `issue_id` and `issue_key` stored for identification
- `project_key` for project context
- `is_processed` set to `false`

JiraRaw → Task:

- `issue_summary` → `task.title`
- `issue_description` → `task.description`
- `issue_status` → `task.status` [13]
- `issue_priority` → `task.priority` [14]
- `issue_due_date` → `task.due_date`
- `issue_created_date` → `task.date_created`
- `jira_issue_id` stored for source tracking
- Marks `jira_raw.is_processed = true`

4.3.3. Edge Cases

- **Multiple Jira sites**: User can connect multiple Jira instances, each as separate integration
- **JQL pagination**: Uses `startAt` and `maxResults` for large result sets
- **Missing fields**: Handles null values gracefully
- **Custom fields**: Currently only maps standard Jira fields
- **Token expiration**: Refresh tokens used to obtain new access tokens

[12] See Glossary

[13] Mapped: "To Do" → "pending", "In Progress" → "in_progress", "Done" → "completed"

[14] Mapped: "Highest" → "high", "Medium" → "medium", "Low" → "low"

5. Duplicate Detection Engine

5.1. How It Works

The duplicate detection engine uses a two-stage approach:

5.1.1. Stage 1: Semantic Similarity (SBERT + pgvector)

1. **Embedding Generation**:
   - Uses SBERT model `all-mpnet-base-v2` (768 dimensions)
   - Combines task title and description: `"{title}. {description}"`
   - Generates vector embedding for each task
   - Stores in `task_embeddings` table with pgvector `Vector(768)` type
2. **Similarity Search**:
   - Uses pgvector cosine similarity operator `<=>`
   - HNSW index enables fast approximate nearest neighbor search
   - Finds tasks with similarity >= threshold (default: 0.7)
   - Groups top-k most similar tasks (default: 7 tasks per group)
3. **Group Creation**:
   - Creates `DuplicateGroup` records with `task_ids` array
   - Task IDs are sorted to ensure consistent representation
   - Groups are mutually exclusive: each task can only belong to one group at a time
   - Before creating a group, the system checks if any task is already in an existing group using the GIN index
   - Unique constraint `(user_id, task_ids)` prevents duplicate groups
   - Initial status: `gemini_status = 'pending'`

5.1.2. Stage 2: LLM Validation (Google Gemini)

1. **Prompt Construction**:
   - Builds detailed prompt with all tasks in group
   - Includes task titles, descriptions, due dates, priorities
   - Asks Gemini to determine if tasks are true duplicates
2. **LLM Call**:
   - Sends prompt to Gemini 2.5 Flash model
   - Requests JSON response with `duplicate_task_ids` array
   - Implements retry logic with exponential backoff
3. **Status Update**:
   - `confirmed` : All tasks are duplicates
   - `rejected` : Tasks are not duplicates
   - `mixed` : Some tasks are duplicates, some are not
   - Updates `duplicate_groups.gemini_status` accordingly

5.2. Why This Approach?

Semantic Understanding: SBERT embeddings capture meaning, not just keywords. Tasks like "Fix login bug" and "Resolve authentication issue" are detected as similar even with different wording.

Two-Stage Validation: Vector similarity is fast but can have false positives. LLM validation adds reasoning to reduce false positives and improve accuracy.

Scalability: pgvector HNSW indexing provides efficient search that scales well, enabling duplicate detection across thousands of tasks.

5.3. Performance Considerations

Batch Processing:
- Embeddings generated in batches for efficiency
- Duplicate detection runs on all tasks, but groups are processed incrementally
- LLM validation processes groups one at a time to manage API rate limits

Indexing:
- HNSW index on `task_embeddings.embedding` column for vector similarity search
- Index created with: `CREATE INDEX ON task_embeddings USING hnsw (embedding vector_cosine_ops)`
- GIN index on `duplicate_groups.task_ids` array column for O(n log n) array containment queries
- Index created with: `CREATE INDEX ON duplicate_groups USING GIN (task_ids)`
- Enables fast similarity search and duplicate group lookups even with large datasets

Caching:
- Embeddings are generated once and reused
- Only tasks with `is_embedded=False` are processed
- Duplicate groups are cached until tasks change

5.4. Future Improvements

- **Incremental Detection**: Only check new tasks against existing tasks
- **Threshold Tuning**: User-configurable similarity thresholds
- **Multi-Model Ensemble**: Combine multiple embedding models for better accuracy
- **Active Learning**: Use user feedback to improve duplicate detection
- **Real-time Detection**: Detect duplicates as tasks are created, not in batch

6. Frontend Architecture

6.1. Streamlit Structure

The frontend is built using Streamlit, a Python-based web framework that enables rapid development of data applications.

6.1.1. Application Structure

```
frontend/
├── app.py                # Main entry point, routing
├── app_pages/            # Page components
│   ├── dashboard.py      # Main dashboard
│   ├── my_tasks.py       # Task list view
│   ├── duplicates.py     # Duplicate management
│   ├── task_metrics.py   # Analytics and metrics
│   └── integrations.py   # Integration management
├── components/           # Reusable UI components
│   ├── auth.py           # Login/register forms
│   ├── sidebar.py        # Navigation sidebar
│   ├── charts.py         # Data visualizations
│   └── (and other component files)
├── services/             # API client services
│   ├── api_client.py     # HTTP client wrapper
│   ├── auth_service.py   # Authentication logic
│   └── (and other service files)
└── utils/                # Helper functions
    ├── session_manager.py  # Manages Streamlit session state
    ├── token_storage.py    # Browser localStorage for token persistence
    └── (and other utility files)
```

6.2. Session Management

Streamlit's session state is used for:

- **Authentication**: Stores `authenticated` flag and user info
- **Token Storage**: Stores session token for API requests
- **Page Navigation**: Tracks current page (`current_page`)
- **UI State**: Form inputs, filters, selected items

Token persistence uses `streamlit-local-storage` to persist authentication across page refreshes.

6.3. API Communication Pattern

All API calls go through the `api_client.py` service:

1. **Request**: Component calls service method (e.g., `task_service.get_tasks()`)
2. **Service**: Service method calls `api_client.request()` with endpoint and method
3. **Client**: `api_client` adds authentication header (session token) and makes HTTP request
4. **Response**: Response parsed and returned to component
5. **Error Handling**: Errors caught and displayed to user via Streamlit error messages

6.4. Website Overview

6.4.1. Task Metrics

- Analytics dashboard displaying task metrics and analytics
- Status summary, comprehensive metrics, and task progress
- Duplicate detection metrics and accuracy
- Slack-specific metrics
- Charts and visualizations for task data

6.4.2. My Tasks

- List view of all user's tasks
- Filtering and sorting capabilities
- Task details: Title, description, due date, source link
- Create new tasks and edit existing tasks
- Mark tasks as complete

6.4.3. Duplicates

- Review duplicate task groups identified by Gemini
- Group details: Tasks in group, validation status
- Actions: Resolve duplicates by selecting original task, remove individual tasks from groups
- Filter by validation status (pending/confirmed/rejected)

6.4.4. Integrations

- Manage platform integrations (Trello, Slack, Jira)
- Connect new integration via OAuth flow
- Disconnect integration
- Sync status and last sync time
- Manual sync trigger for each integration
- Jira workspace selection for multiple sites

7. API Reference

7.1. Authentication Endpoints

7.1.1. POST /login

Request:

```
{
  "email": "user@example.com",
  "password": "password123"
}
```

Response:

```
{
  "message": "Login successful",
  "data": {
    "session_token": "abc123...",
    "user": {
      "id": 1,
      "email": "user@example.com",
      "first_name": "John",
      "last_name": "Doe"
    }
  }
}
```

7.1.2. POST /register

Request:

```
{
  "email": "user@example.com",
  "password": "password123",
  "first_name": "John",
  "last_name": "Doe"
}
```

Response:

```
{
  "message": "Registration successful",
  "data": {
    "session_token": "abc123...",
    "user": { ... }
  }
}
```

7.2. Integration Endpoints

7.2.1. GET /jira/auth/connect

Initiates Jira OAuth flow. Returns OAuth URL for user to visit.

Response:

```
{
  "message": "Jira OAuth URL retrieved successfully",
  "data": {
    "oauth_url": "https://auth.atlassian.com/authorize?...",
```

```
    "state": "csrf_token_here"
  }
}
```


7.2.2. POST /jira/sync

Syncs Jira issues for authenticated user.

Query Parameters:

- `project_key` (optional): Sync specific project only

Response:


```
{
  "message": "Jira sync completed",
  "data": {
    "issues_synced": 42,
    "tasks_created": 38
  }
}
```


Similar endpoints exist for `/trello/sync` and `/slack/sync`.

7.3. Task Endpoints

7.3.1. GET /tasks

Retrieves all tasks for authenticated user.

Query Parameters:

- `status` (optional): Filter by status
- `source` (optional): Filter by source (slack/trello/jira)
- `limit` (optional): Limit number of results

Response:


```
{
  "message": "Tasks retrieved successfully",
  "data": [
    {
      "id": 1,
      "title": "Fix login bug",
      "description": "Fix the login bug",
      "status": "pending",
      "priority": "high",
      "source": "slack",
      "due_date": "2025-12-31T00:00:00",
      "date_created": "2025-11-15T10:00:00"
    },
    ...
  ]
}
```


7.3.2. POST /tasks/embed

Generates embeddings for tasks that don't have them yet.

Response:

```
{
  "message": "Embeddings generated successfully",
  "data": {
    "tasks_embedded": 15
  }
}
```


7.4. Duplicate Endpoints

7.4.1. GET /duplicates

Retrieves duplicate groups for authenticated user.

Query Parameters:

- `status` (optional): Filter by gemini_status (pending/confirmed/rejected)

Response:


```
{
  "message": "Duplicate groups retrieved successfully",
  "data": [
    {
      "id": 1,
      "task_ids": [1, 5, 12],
      "gemini_status": "confirmed",
      "tasks": [
        {
          "id": 1,
          "title": "Fix login bug",
          ...
        },
        ...
      ]
    },
    ...
  ]
}
```


7.4.2. POST /duplicates/resolve

Resolves a duplicate group by marking tasks as duplicates.

Request:


```
{
  "group_id": 1,
  "primary_task_id": 1,
  "duplicate_task_ids": [5, 12]
}
```


Response:


```
{
  "message": "Duplicates resolved successfully",
  "data": {
    "tasks_marked_duplicate": 2
  }
}
```

7.4.3. POST /duplicates/detect

Triggers duplicate detection for user's tasks.

Response:


```
{
  "message": "Duplicate detection completed",
  "data": {
    "groups_created": 8,
    "tasks_processed": 150
  }
}
```


7.4.4. POST /duplicates/sync

Triggers Big Sync: comprehensive synchronization that syncs all integrations, generates embeddings, detects duplicates, and validates with Gemini in a single operation.

Response:


```
{
  "success": true,
  "summary": {
    "integrations_synced": ["slack", "trello", "jira"],
    "tasks_ingested": 150,
    "tasks_embedded": 145,
    "duplicate_groups_created": 8,
    "gemini_verified_groups": 8
  }
}
```

8. Development Guide

8.1. Local Setup

8.1.1. Prerequisites

- Docker and Docker Compose installed
- Git for version control
- Text editor or IDE

8.1.2. Environment Variables

Create a `.env` file in the project root:

```
# Database
DATABASE_URL=postgresql://postgres:123456@db:5432/unsprawl

# Google Gemini
GEMINI_API_KEY=your_gemini_api_key_here

# Trello OAuth
TRELLO_CLIENT_ID=your_trello_client_id
TRELLO_CLIENT_SECRET=your_trello_client_secret

# Slack OAuth
SLACK_CLIENT_ID=your_slack_client_id
SLACK_CLIENT_SECRET=your_slack_client_secret
SLACK_REDIRECT_URI=http://localhost:8000/slack/auth/callback

# Jira OAuth
JIRA_CLIENT_ID=your_jira_client_id
JIRA_CLIENT_SECRET=your_jira_client_secret
JIRA_REDIRECT_URI=http://localhost:8000/jira/auth/callback
```

8.1.3. Docker Commands

Start all services:

```
docker compose up --build
```

Run database migrations:

```
docker compose run --rm migrate
```

Stop services:

```
docker compose down
```

View logs:

```
docker compose logs -f backend
```

8.1.4. Running the Frontend

The frontend is a Streamlit application that runs separately from Docker:

```
# From the project root directory
streamlit run frontend/app.py
```

The frontend will be available at `http://localhost:8501` by default. Make sure the backend is running in Docker before starting the frontend.

8.1.5. Running Migrations

Migrations are managed with Alembic and should be run through Docker since the database is hosted in a Docker container:

```
# Apply all pending migrations
docker compose run --rm migrate

# Create a new migration (requires backend service to be running)
docker compose exec backend alembic revision --autogenerate -m "description of changes"

# Apply migrations
docker compose exec backend alembic upgrade head

# Rollback last migration
docker compose exec backend alembic downgrade -1

# View migration history
docker compose exec backend alembic history

# Show current migration version
docker compose exec backend alembic current
```

Note: The `docker compose run --rm migrate` command uses a dedicated migration service defined in `docker-compose.yml`. For other Alembic commands, use `docker compose exec backend` to run commands inside the running backend container.

8.2. Adding a New Integration

To add a new integration (e.g., Asana, Monday.com):

1. **Create Raw Data Model**: Add new model in `backend/models/raw_data.py` (e.g., `AsanaRaw`)
2. **Create Service**: Add service in `backend/services/` (e.g., `asana_service.py`)
   - Implement OAuth flow
   - Implement data sync logic
   - Implement raw-to-task transformation
3. **Create Router**: Add router in `backend/api/` (e.g., `asana_router.py`)
   - OAuth endpoints: `/auth/connect`, `/auth/callback`
   - Sync endpoint: `/sync`
4. **Register Router**: Add to `backend/app.py`:

   ```
   from api.asana_router import router as asana_router
   app.include_router(asana_router, prefix="/asana", tags=["Asana"])
   ```

5. **Create Repository**: Add repository in `backend/repositories/` if needed
6. **Update Frontend**: Add integration card in `frontend/app_pages/integrations.py`
7. **Run Migration**: Create and run Alembic migration for new raw data table

9. Deployment

9.1. Current Deployment

Unsprawl is currently deployed on Digital Ocean App Platform:
- **Backend**: One container running the FastAPI application
- **Frontend**: One container running the Streamlit application
- **Database**: Supabase (managed PostgreSQL with pgvector extension)

9.2. Production Considerations

9.2.1. Database Hosting

pgvector Requirement: Database must support pgvector extension. Options:
- Supabase with pgvector extension
- Self-hosted PostgreSQL with pgvector installed
- AWS RDS with pgvector extension

9.2.2. Environment Configuration

Secrets Management:
- Use environment variables (not hardcoded)
- Use secret management service (e.g., Digital Ocean Secrets, AWS Secrets Manager)
- Rotate API keys regularly

Configuration:
- Set `DATABASE_URL` to production database
- Configure OAuth redirect URIs for production domain
- Set appropriate CORS origins (not `*`)

9.2.3. Scaling Considerations

Backend:
- FastAPI supports async, can handle many concurrent requests
- Consider horizontal scaling with load balancer
- Use connection pooling for database

Database:
- Monitor query performance
- Add indexes as needed

Embedding Generation:
- Currently CPU-bound (SBERT model)
- Consider GPU acceleration for large-scale deployments
- Batch processing to optimize throughput

LLM Calls:
- Implement rate limiting to manage API costs
- Cache LLM responses where possible
- Consider async batch processing

10. Known Issues & Future Work

10.1. Current Limitations

Performance:
- Duplicate detection runs on all tasks (not incremental)
- Embedding generation is CPU-bound and can be slow for large datasets
- LLM validation processes groups sequentially

Functionality:
- No automatic token refresh for Jira (manual re-authentication required)
- Custom Jira fields not mapped to tasks
- No support for task comments from source platforms
- Limited task filtering options in frontend

User Experience:
- No email notifications for duplicate detection
- No task assignment features
- Limited analytics and reporting

10.2. Potential Improvements

ML/AI:
- Fine-tune SBERT model on task-specific data
- Implement active learning from user feedback
- Custom LLM prompts per integration type

User Experience:
- Real-time duplicate detection as tasks are created
- Slack Task Creation: Automatically create tasks from Slack messages
- Smart task suggestions based on user history
- Integration health monitoring and alerts
- Customizable duplicate detection thresholds

10.3. Tech Debt

Code Quality:
- Add unit and integration tests
- More extensive documentation

Infrastructure:
- Set up CI/CD pipeline
- Implement proper logging and monitoring
- Add database migration testing
- Implement staging environment

11. Appendix

11.1. Glossary of Terms

SBERT: A framework for generating sentence embeddings using transformer models.

pgvector: PostgreSQL extension for storing and querying vector embeddings.

HNSW: Hierarchical Navigable Small World, an approximate nearest neighbor search algorithm used by pgvector.

Cosine Similarity: A measure of similarity between two vectors, calculated as the cosine of the angle between them. Range: −1 to 1, where 1 means identical.

OAuth 2.0: An authorization framework that allows applications to obtain limited access to user accounts on external services.

Embedding: A numerical representation of text (or other data) as a vector in high-dimensional space, capturing semantic meaning.

LLM Judge: Using a Large Language Model to validate or make decisions, in this case validating whether tasks are duplicates.

JQL: Jira Query Language, used to search for issues in Jira.

DeBERTa-v3: A pre-trained language model that can understand and classify text. In Unsprawl, it reads Slack messages and determines whether they contain actionable tasks (like "Fix the login bug") or are just casual conversation (like "Thanks for the update"). The model works without needing training on Slack-specific data.

Session Token: A unique identifier issued by the backend after user authentication, stored in the `sessions` table and used to authenticate subsequent API requests. The frontend includes this token in request headers.

11.2. External Documentation Links

APIs:
- Jira REST API: https://developer.atlassian.com/cloud/jira/platform/rest/v3/°
- Trello API: https://developer.atlassian.com/cloud/trello/guides/rest-api/api-reference/°
- Slack API: https://api.slack.com/°

Libraries & Tools:
- FastAPI: https://fastapi.tiangolo.com/°
- pgvector: https://github.com/pgvector/pgvector°
- Sentence Transformers: https://www.sbert.net/°
- Google Gemini API: https://ai.google.dev/docs°
- Streamlit: https://docs.streamlit.io/°
- SQLAlchemy: https://docs.sqlalchemy.org/°
- Alembic: https://alembic.sqlalchemy.org/°
- N-tier architecture: https://learn.microsoft.com/en-us/azure/architecture/guide/architecture-styles/n-tier°
- Digital Ocean: https://docs.digitalocean.com/products/app-platform/°

11.3. Team Contacts

Project Lead: Kaydence Lin lin.kay@northeastern.edu

Technical Lead: Jerome Rodrigo rodrigo.j@northeastern.edu

Team Members:

- Gordon Bie - bie.g@northeastern.edu
- Deekshita Madhalam - madhalam.d@northeastern.edu
- Eleanor Meltzer - meltzer.el@northeastern.edu
- Gokul Ramanan - ramanan.g@northeastern.edu
- Vichu Selvaraju - selvaraju.v@northeastern.edu
- Melina Yang - yang.melina@northeastern.edu