Jira AI Assistant - Requirements Document

AI-Powered Intelligent Ticket Management System

Document Version: 1.0
Date: December 2024
Status: Prototype Requirements
Author: Development Team

Table of Contents

---

1. Executive Summary

1.1 Project Overview

Project Name: Jira AI Assistant - Intelligent Ticket Management System

Objective: Develop an AI-powered automation system that transforms natural language prompts into fully-managed Jira tickets with automatic estimation, intelligent breakdown, and smart assignment based on team capacity and priority.

Prototype Scope: MVP (Minimum Viable Product) focusing on core automation flows with learning capabilities.

1.2 Key Features

- ✅ Natural language story creation
- ✅ AI-powered estimation using historical data
- ✅ Automatic story breakdown into subtasks
- ✅ Intelligent assignment (priority + bandwidth)
- ✅ Real-time capacity tracking
- ✅ Feedback learning loop
- ✅ Multi-channel access (Web, Slack, API)

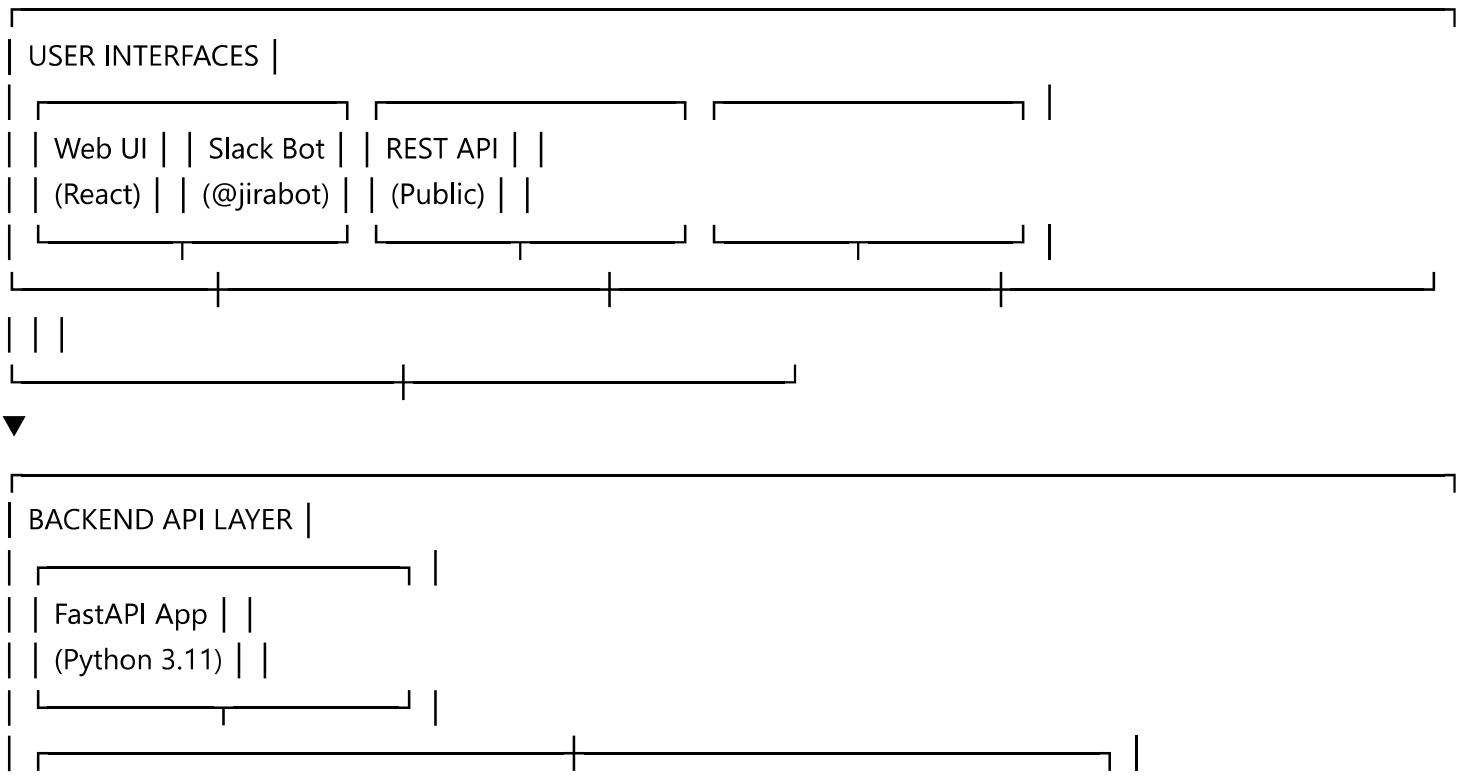
1.3 Business Value

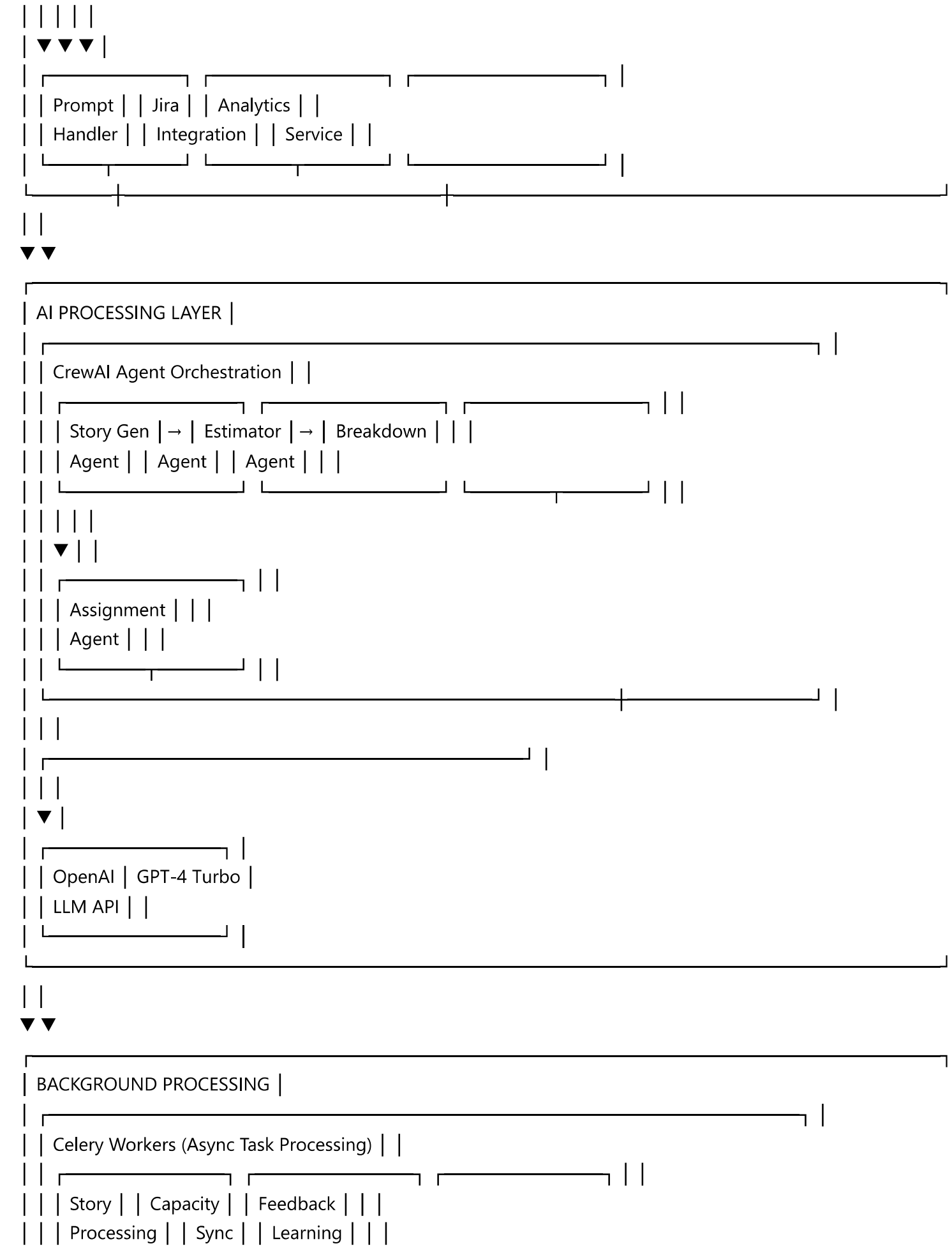
Benefit	Impact
Time Savings	70% reduction in story creation time
Estimation Accuracy	75% accuracy improvement
Workload Balance	Automated capacity-based distribution
Team Productivity	Reduced manual assignment overhead
Knowledge Retention	AI learns from team decisions

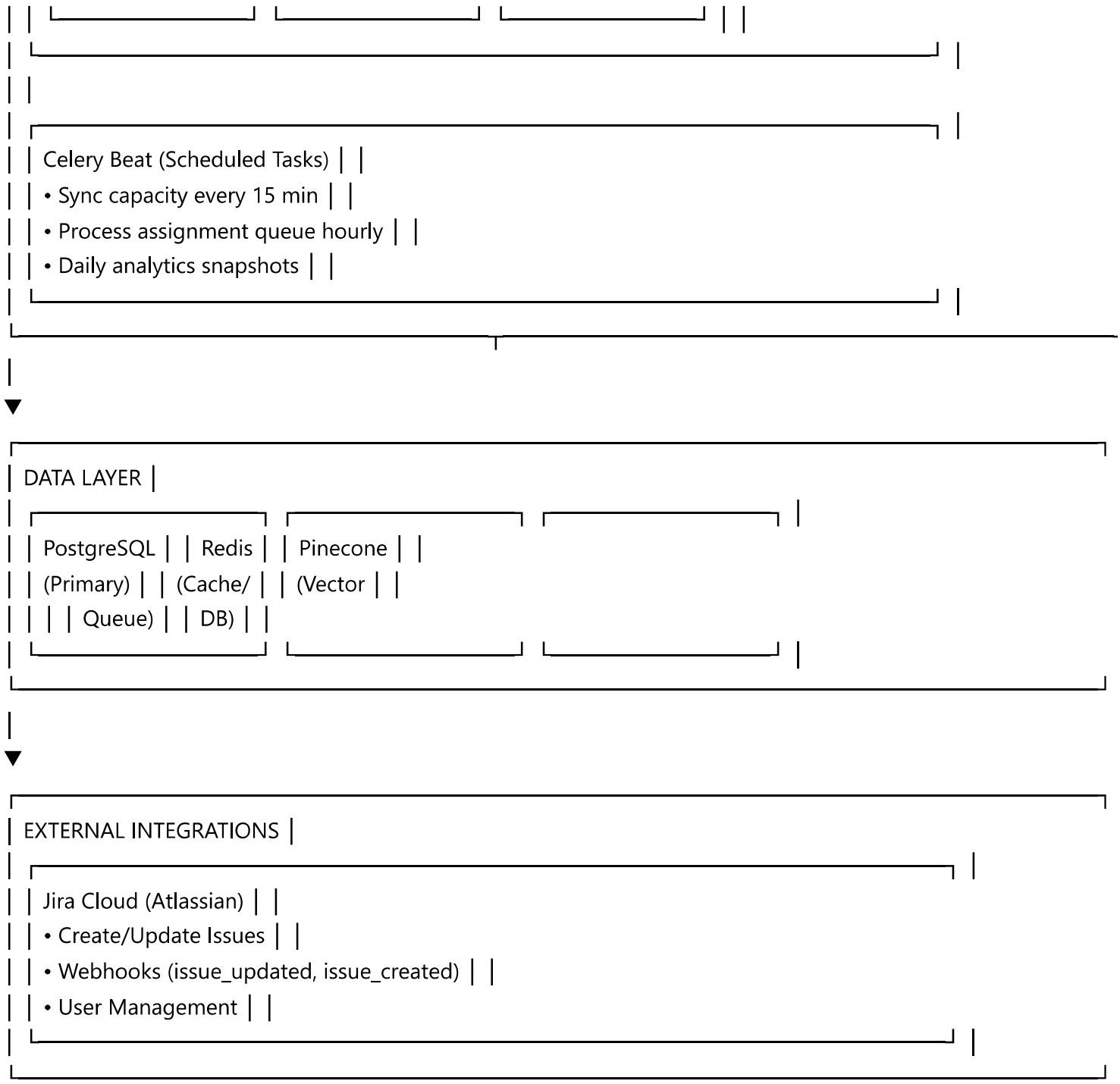
---

2. System Architecture

2.1 High-Level Architecture




Prompt Handler
Jira Integration
Analytics Service
AI PROCESSING LAYER
CrewAI Agent Orchestration
Story Gen Agent
Estimator Agent
Breakdown Agent
Assignment Agent
OpenAI GPT-4 Turbo
LLM API
BACKGROUND PROCESSING
Celery Workers (Async Task Processing)
Story Processing
Capacity Sync
Feedback Learning



2.2 Component Descriptions

2.2.1 User Interface Layer

- **Web UI (React)**: Primary interface for story creation and monitoring
- **Slack Bot**: Conversational interface for quick story creation
- **REST API**: Programmatic access for integrations

2.2.2 Backend API Layer

- **FastAPI**: High-performance async Python web framework
- **Prompt Handler**: Processes natural language inputs
- **Jira Integration**: Bidirectional sync with Jira Cloud
- **Analytics Service**: Performance metrics and insights

2.2.3 AI Processing Layer

- **CrewAI Orchestration**: Multi-agent coordination
- **Story Generator Agent**: Creates structured stories from prompts
- **Estimator Agent**: Calculates story points using RAG
- **Breakdown Agent**: Splits large stories into subtasks
- **Assignment Agent**: Matches tickets to team members

2.2.4 Background Processing

- **Celery Workers**: Async task execution
- **Celery Beat**: Scheduled periodic tasks
- **Task Types**: Story processing, capacity sync, learning updates

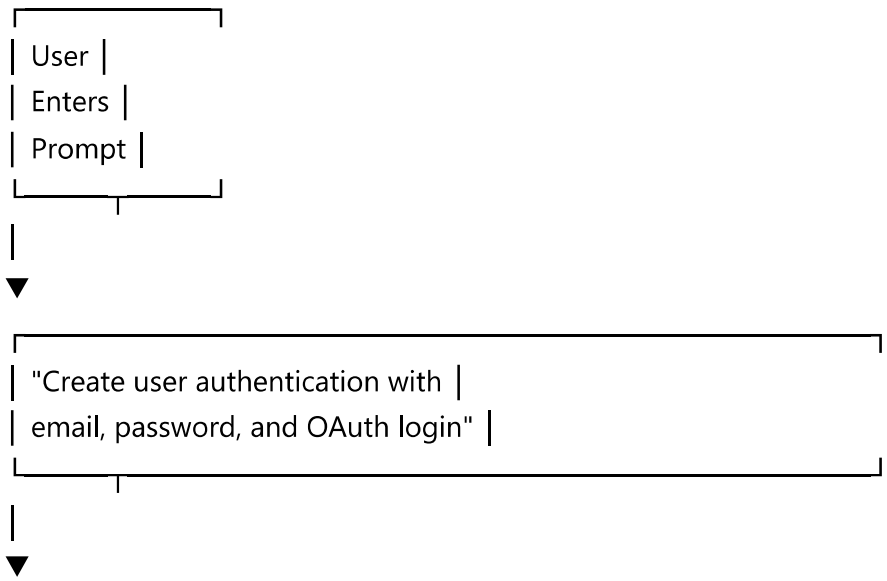
2.2.5 Data Layer

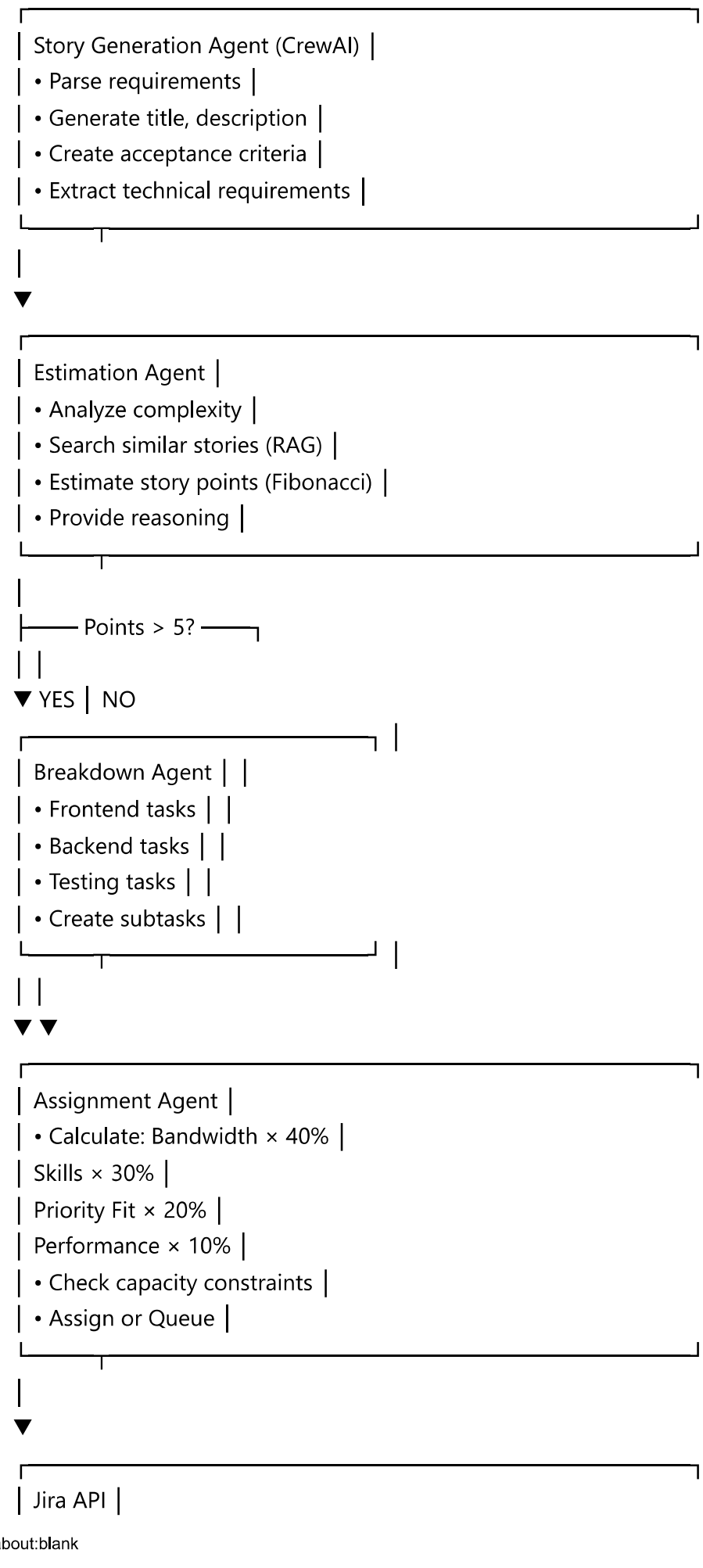
- **PostgreSQL**: Relational data (stories, team, feedback)
- **Redis**: Cache, message broker, session storage
- **Pinecone/ChromaDB**: Vector embeddings for RAG

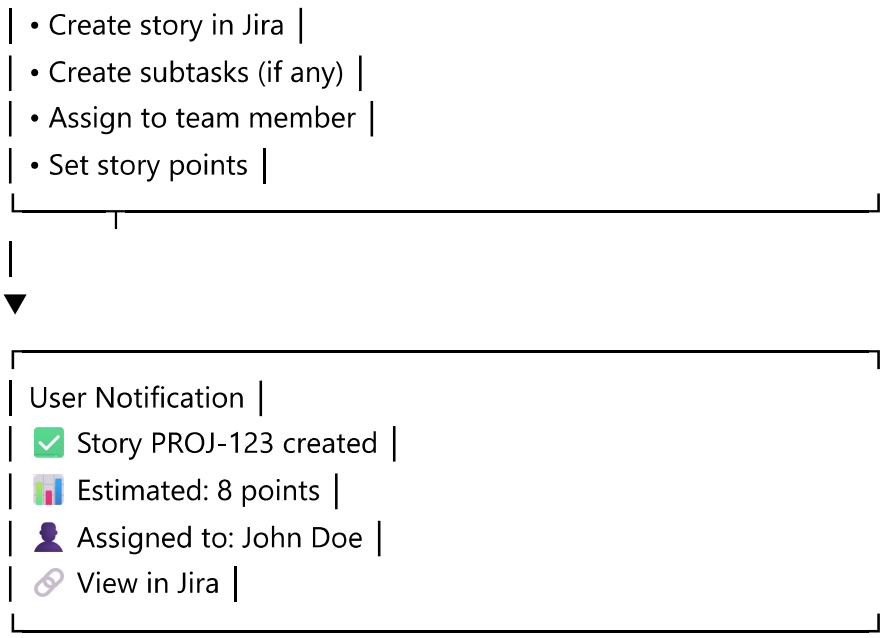
---

3. Core User Flows

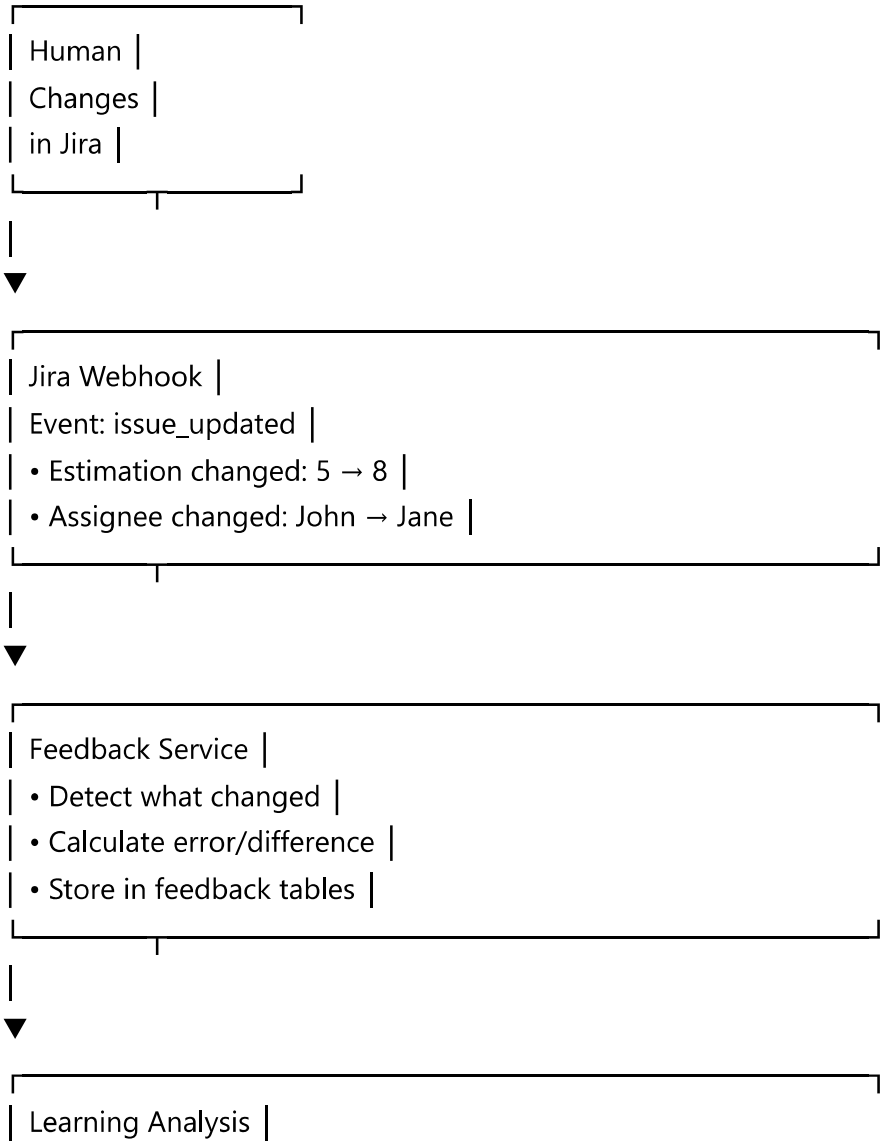
3.1 Story Creation Flow

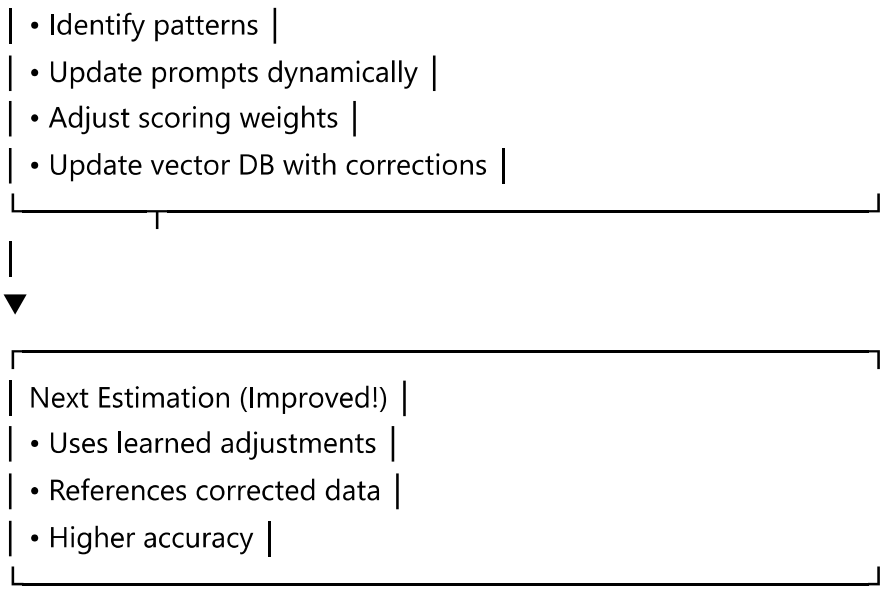



Story Generation Agent (CrewAI)
• Parse requirements
• Generate title, description
• Create acceptance criteria
• Extract technical requirements
Estimation Agent
• Analyze complexity
• Search similar stories (RAG)
• Estimate story points (Fibonacci)
• Provide reasoning
Points > 5?
YES
NO
Breakdown Agent
• Frontend tasks
• Backend tasks
• Testing tasks
• Create subtasks
Assignment Agent
• Calculate: Bandwidth × 40%
Skills × 30%
Priority Fit × 20%
Performance × 10%
• Check capacity constraints
• Assign or Queue
Jira API

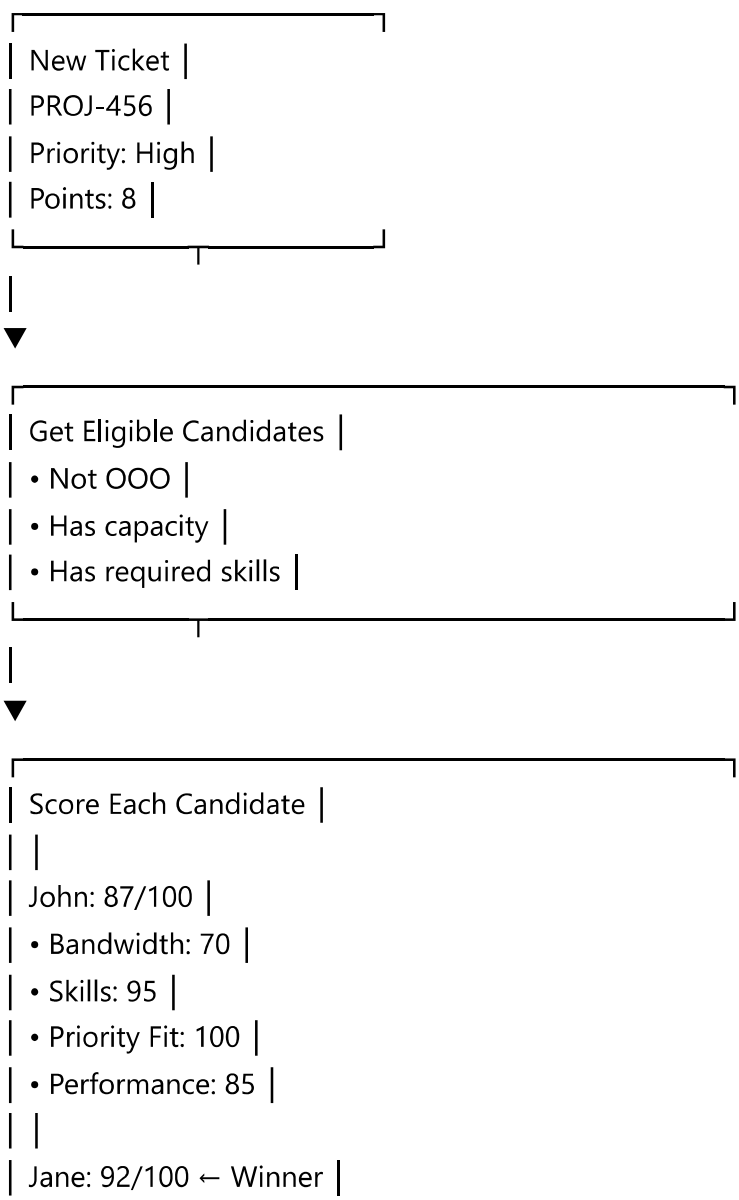


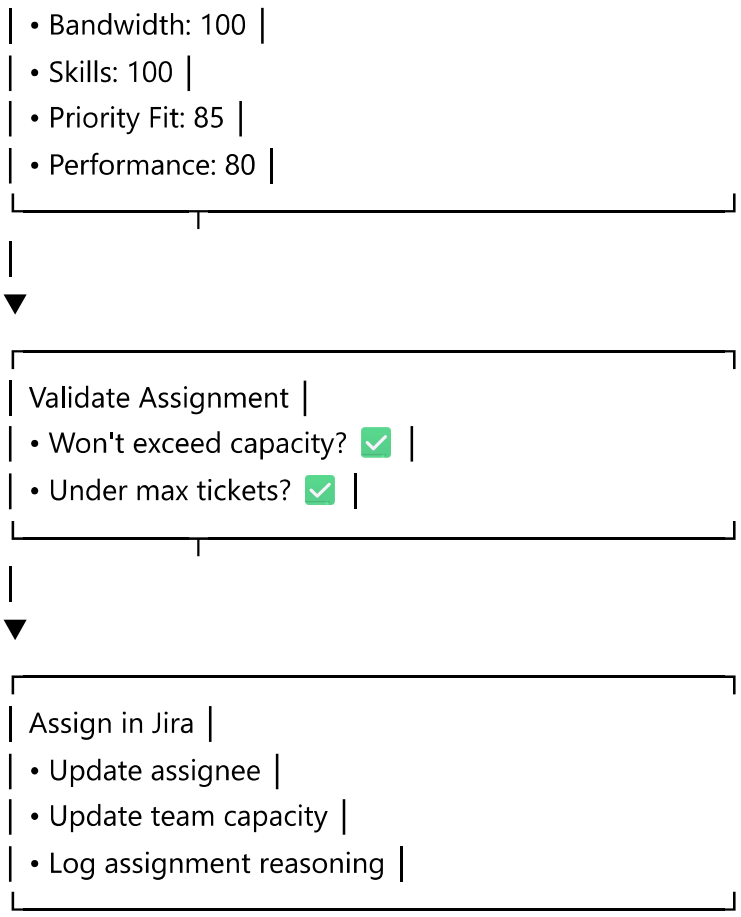
3.2 Feedback Learning Loop





3.3 Assignment Decision Flow





---

4. Functional Requirements

4.1 Natural Language Story Creation

ID	Requirement	Priority	Acceptance Criteria
FR-1.1	Accept natural language prompt (min 10 chars)	High	System accepts prompts ≥10 characters
FR-1.2	Generate story title in user story format	High	Title follows "As a [user], I want [feature]"
FR-1.3	Create detailed description with context	High	Description includes problem, solution, impact
FR-1.4	Generate 3-7 acceptance criteria	High	List of testable criteria provided
FR-1.5	Extract technical requirements automatically	Medium	Technical notes section populated

ID	Requirement	Priority	Acceptance Criteria
FR-1.6	Identify required skills from description	Medium	Skills array returned (e.g., ["Python", "React"])

4.2 AI-Powered Estimation

ID	Requirement	Priority	Acceptance Criteria
FR-2.1	Estimate using Fibonacci scale (1,2,3,5,8,13,21)	High	Only valid Fibonacci numbers returned
FR-2.2	Use RAG to find similar historical stories	High	Queries vector DB with 5 similar stories
FR-2.3	Provide estimation reasoning	Medium	Text explanation of estimation logic
FR-2.4	Include confidence score	Low	Percentage confidence (e.g., 75%)

4.3 Intelligent Story Breakdown

ID	Requirement	Priority	Acceptance Criteria
FR-3.1	Auto-break stories > 5 points into subtasks	High	Triggered when estimated_points > 5
FR-3.2	Create 4-8 subtasks per story	High	Number of subtasks in range
FR-3.3	Categorize tasks (Frontend, Backend, Testing, etc.)	Medium	Each subtask has category label
FR-3.4	Link subtasks to parent story in Jira	High	Parent-child relationship created

4.4 Smart Assignment (Priority + Bandwidth)

ID	Requirement	Priority	Acceptance Criteria
FR-4.1	Calculate assignment score based on 4 factors	High	Score = Bandwidth×40% + Skills×30% + Priority×20% + Performance×10%
FR-4.2	Never exceed team member max capacity	High	Assignment rejected if over capacity
FR-4.3	Assign high priority to senior developers	High	Priority="High" filters out Junior devs
FR-4.4	Queue tickets when no capacity available	Medium	Ticket added to assignment queue

ID	Requirement	Priority	Acceptance Criteria
FR-4.5	Respect out-of-office status	High	OOO members excluded from candidates
FR-4.6	Balance workload across team	Medium	No member gets >30% more work than average

4.5 Capacity Management

ID	Requirement	Priority	Acceptance Criteria
FR-5.1	Sync capacity from Jira every 15 minutes	High	Celery beat task runs on schedule
FR-5.2	Track current story points per developer	High	Database field updated on assignment
FR-5.3	Track concurrent ticket count	High	Count of open tickets per member
FR-5.4	Calculate availability percentage	Medium	(Available / Max Capacity) × 100
FR-5.5	Mark members as OOO	Medium	API endpoint to set OOO dates

4.6 Feedback & Learning

ID	Requirement	Priority	Acceptance Criteria
FR-6.1	Capture manual estimation changes via webhook	High	Webhook handler processes issue_updated
FR-6.2	Capture manual assignment changes	High	Reassignment logged in FeedbackAssignment table
FR-6.3	Store actual completion time	Medium	Calculated from created → resolved dates
FR-6.4	Update vector DB with completed stories	High	Embeddings added on story completion
FR-6.5	Adjust AI prompts based on patterns	Medium	Dynamic prompt adjustments applied
FR-6.6	Show estimation accuracy metrics	Low	Dashboard displays accuracy percentage

4.7 Multi-Channel Access

ID	Requirement	Priority	Acceptance Criteria
FR-7.1	Web UI for story creation	High	React app with form interface
FR-7.2	Slack bot integration	Medium	Bot responds to mentions and commands
FR-7.3	REST API for programmatic access	High	OpenAPI spec with all endpoints
FR-7.4	Real-time status updates	Low	WebSocket or polling for status

---

5. Non-Functional Requirements

5.1 Performance

ID	Requirement	Target	Measurement Method
NFR-1.1	Story creation end-to-end time	< 10 seconds	API response time logging
NFR-1.2	API response time (95th percentile)	< 2 seconds	APM monitoring
NFR-1.3	Capacity sync completion time	< 30 seconds for 50 users	Celery task duration
NFR-1.4	Concurrent users supported	100 users	Load testing
NFR-1.5	Database query response time	< 500ms	Query logging

5.2 Reliability

ID	Requirement	Target	Measurement Method
NFR-2.1	System uptime	99.5%	Uptime monitoring
NFR-2.2	Webhook delivery retry	3 attempts with exponential backoff	Retry logic implementation
NFR-2.3	Queue durability	Persist failed assignments	Redis persistence enabled
NFR-2.4	Data backup	Daily automated backups	Backup script scheduled
NFR-2.5	Error recovery	Graceful degradation on AI failure	Fallback to default values

5.3 Security

ID	Requirement	Implementation
NFR-3.1	API authentication	JWT tokens or API keys
NFR-3.2	Jira credentials storage	Encrypted environment variables
NFR-3.3	Transport security	HTTPS only (production)
NFR-3.4	Rate limiting	100 requests/minute per user
NFR-3.5	Input validation	Pydantic models for all inputs
NFR-3.6	SQL injection prevention	ORM (SQLAlchemy) parameterized queries

5.4 Scalability

ID	Requirement	Target
NFR-4.1	Team size supported	Up to 50 members (prototype)
NFR-4.2	Story creations per day	500 stories/day
NFR-4.3	Historical data retention	6 months
NFR-4.4	Database growth	Plan for 10GB in 6 months
NFR-4.5	Horizontal scaling	Support multiple worker instances

5.5 Usability

ID	Requirement	Acceptance Criteria
NFR-5.1	Prompt input simplicity	No special syntax required
NFR-5.2	Assignment reasoning visibility	Clear explanation shown to users
NFR-5.3	Error message quality	Actionable error messages
NFR-5.4	Dashboard load time	< 3 seconds
NFR-5.5	Mobile responsiveness	UI works on tablets (optional for prototype)

---

6. Technology Stack

6.1 Backend Technologies

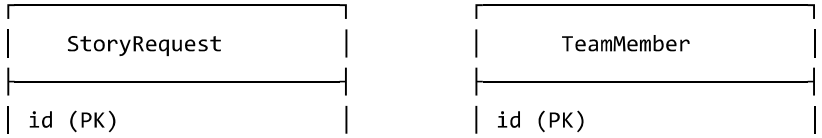
```
 Programming Language: Python 3.11
Web Framework: FastAPI 0.104+
  - Async/await support
  - Automatic OpenAPI documentation
  - Pydantic validation
  - High performance (ASGI)

Agent Framework: CrewAI 0.1+
  - Built on LangChain
  - Multi-agent orchestration
  - Role-based agent design
  - Task delegation support

Async Task Processing: Celery 5.3+
  - Redis broker
  - Flower monitoring UI
  - Beat scheduler for periodic tasks
  - Result backend for task tracking

ORM & Database: SQLAlchemy 2.0+
  - Async support
  - Alembic for migrations
  - PostgreSQL adapter (psycopg2)

LLM Provider: OpenAI
  - Model: GPT-4 Turbo (gpt-4-turbo-preview)
  - API: openai>=1.3.0
  - Embeddings: text-embedding-3-small

Alternative LLM: Anthropic Claude 3
  - Model: claude-3-opus-20240229
  - API: anthropic>=0.7.0
  - Use case: Fallback or comparison

Vector Database Options:
  Primary (Cloud): Pinecone
    - Managed service
    - Auto-scaling
    - Client: pinecone-client>=2.2.4

  Alternative (Self-hosted): ChromaDB
    - Docker container
    - Local development
    - Client: chromadb>=0.4.18

RAG Framework: LangChain
  - Vector store integration
  - Document loaders
  - Embeddings wrapper
  - Version: langchain>=0.0.335
```

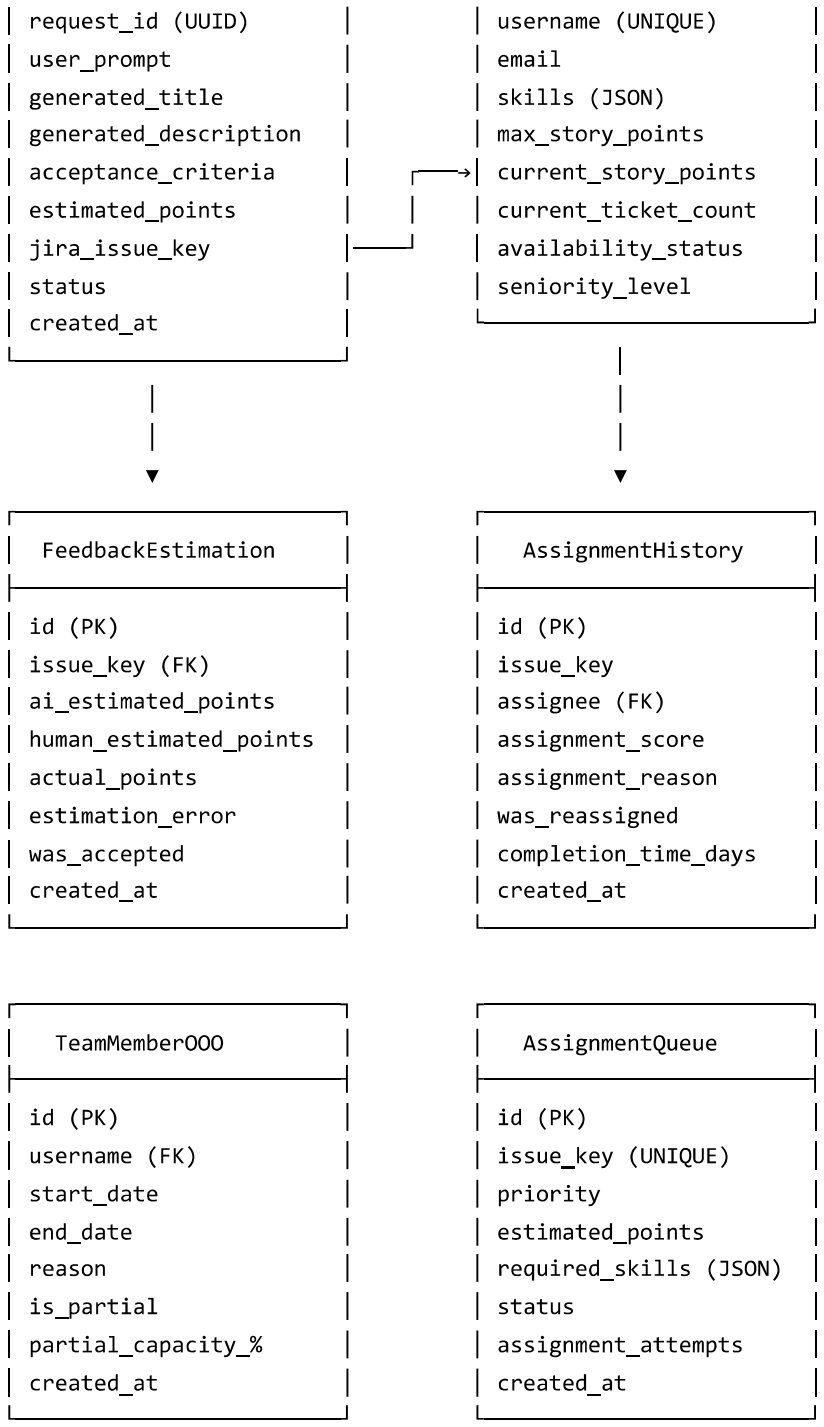
```
Primary Database: PostgreSQL 15
  - Relational data storage
  - JSON/JSONB support
  - Full-text search
  - Connection pooling

Cache & Message Broker: Redis 7
  - Celery broker
  - Session storage
  - Rate limiting
  - Pub/Sub for real-time updates

Vector Storage: Pinecone / ChromaDB
  - Story embeddings
  - Similarity search
  - RAG context retrieval

Framework: React 18
  - Functional components
  - Hooks API
  - Context for state management

Build Tool: Vite
  - Fast HMR (Hot Module Replacement)
  - Optimized production builds
  - ESM support

Styling: Tailwind CSS 3
  - Utility-first CSS
  - Responsive design
  - Custom components

HTTP Client: Axios
  - Promise-based requests
  - Interceptors for auth
  - Request/response transformation

UI Components (Optional):
  - Headless UI (Tailwind official)
  - React Icons
  - Recharts (for analytics)

Jira Integration:
  Library: jira-python (jira>=3.5.2)
  Alternative: atlassian-python-api>=3.41.0
  Authentication: API Token (Basic Auth)
  Features:
    - Issue CRUD operations
    - Custom fields access
    - Webhook registration
    - User management
```

```
    - JQL search

Slack Integration:
  Library: slack-bolt>=1.18.0
  Mode: Socket Mode (no public URL needed)
  Features:
    - App mentions
    - Slash commands
    - Interactive components
    - Message formatting

Monitoring:
  Celery: Flower (web-based monitoring)
  Logging: Python logging module
  Metrics (Future): Prometheus + Grafana

Containerization:
  - Docker 24+
  - Docker Compose 2.20+
  - Multi-stage builds
  - Health checks

Development:
  - python-dotenv for environment management
  - pytest for testing
  - black for code formatting
  - flake8 for linting

Version Control:
  - Git
  - GitHub / GitLab

CI/CD (Future):
  - GitHub Actions
  - Docker Hub / Container Registry
  - Automated testing
  - Deployment pipelines

Production (Future):
  - AWS ECS / EKS
  - GCP Cloud Run
  - Azure Container Instances
  - Managed databases (RDS, Cloud SQL)
```

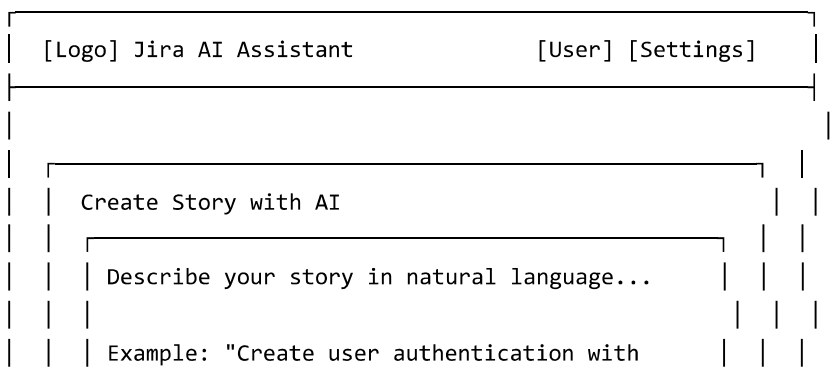
7. Data Models

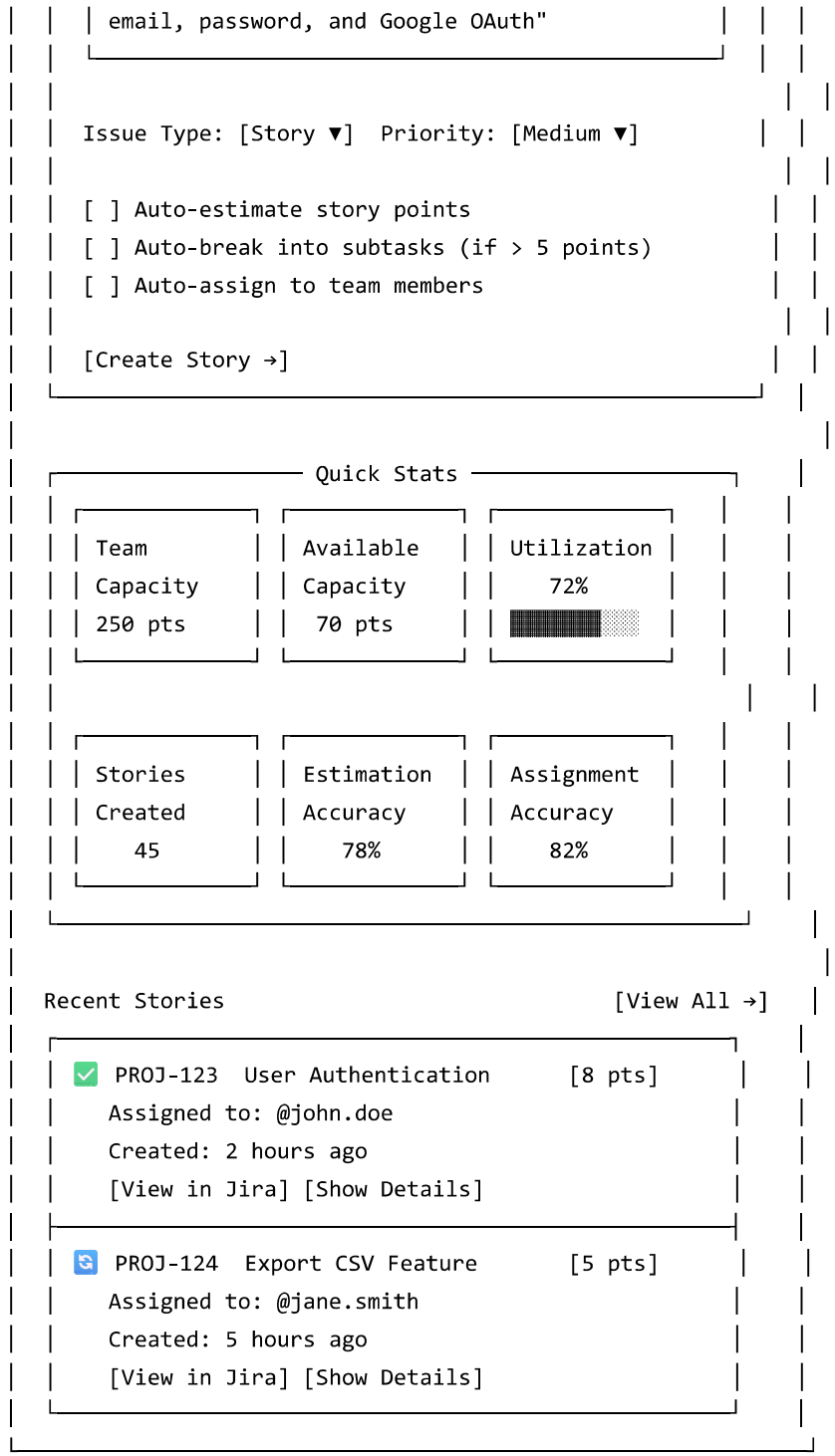
7.1 Entity Relationship Diagram



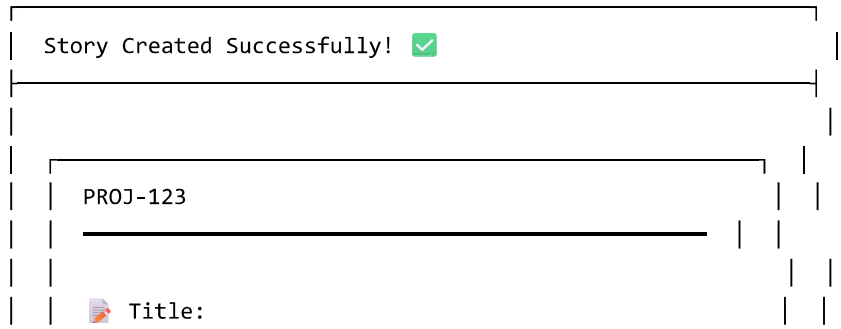


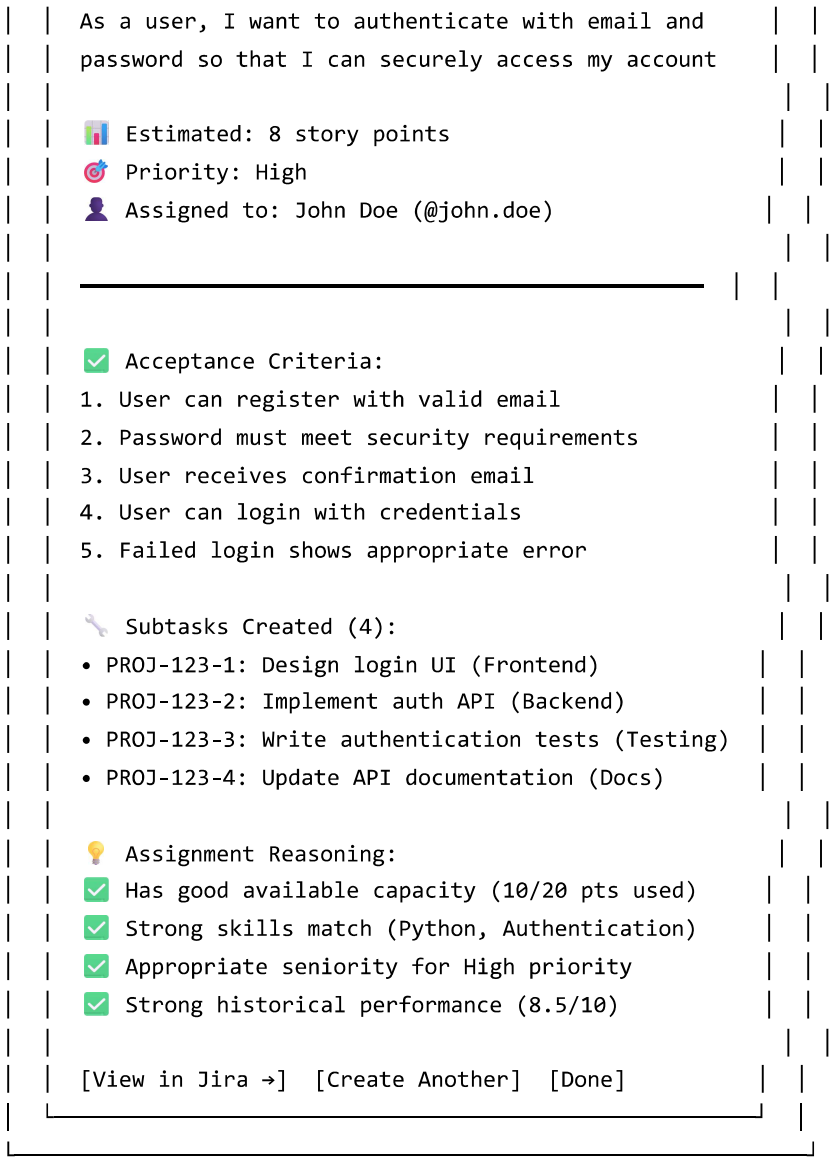
User Interface



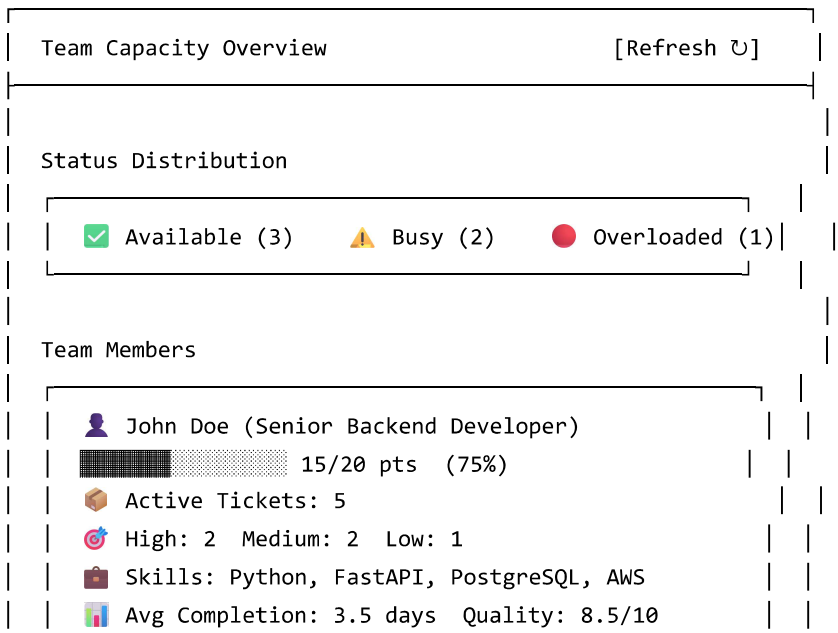


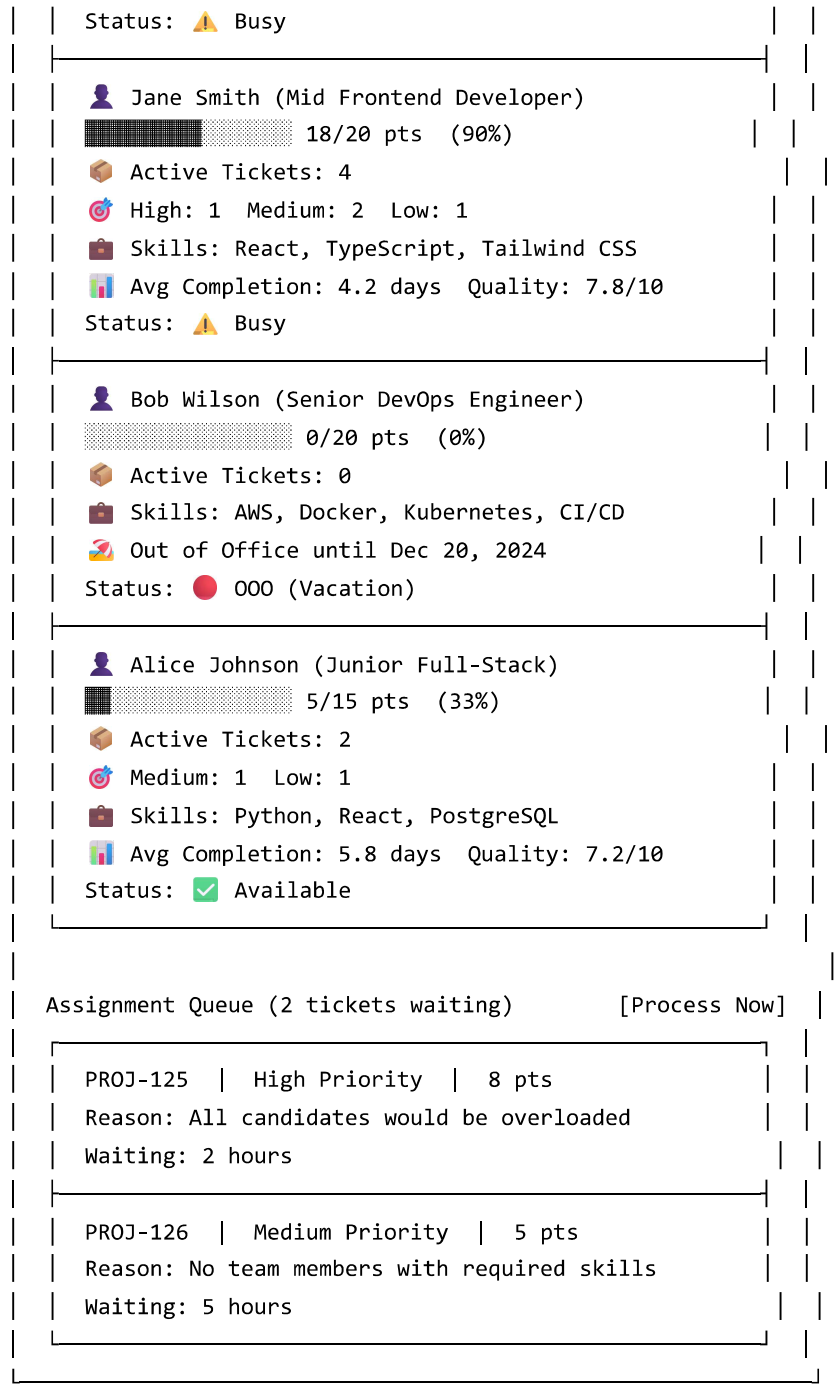
Story Creation Result Screen



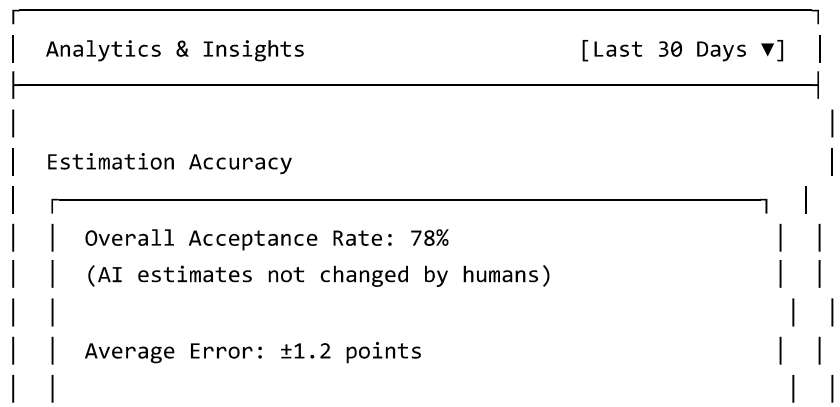


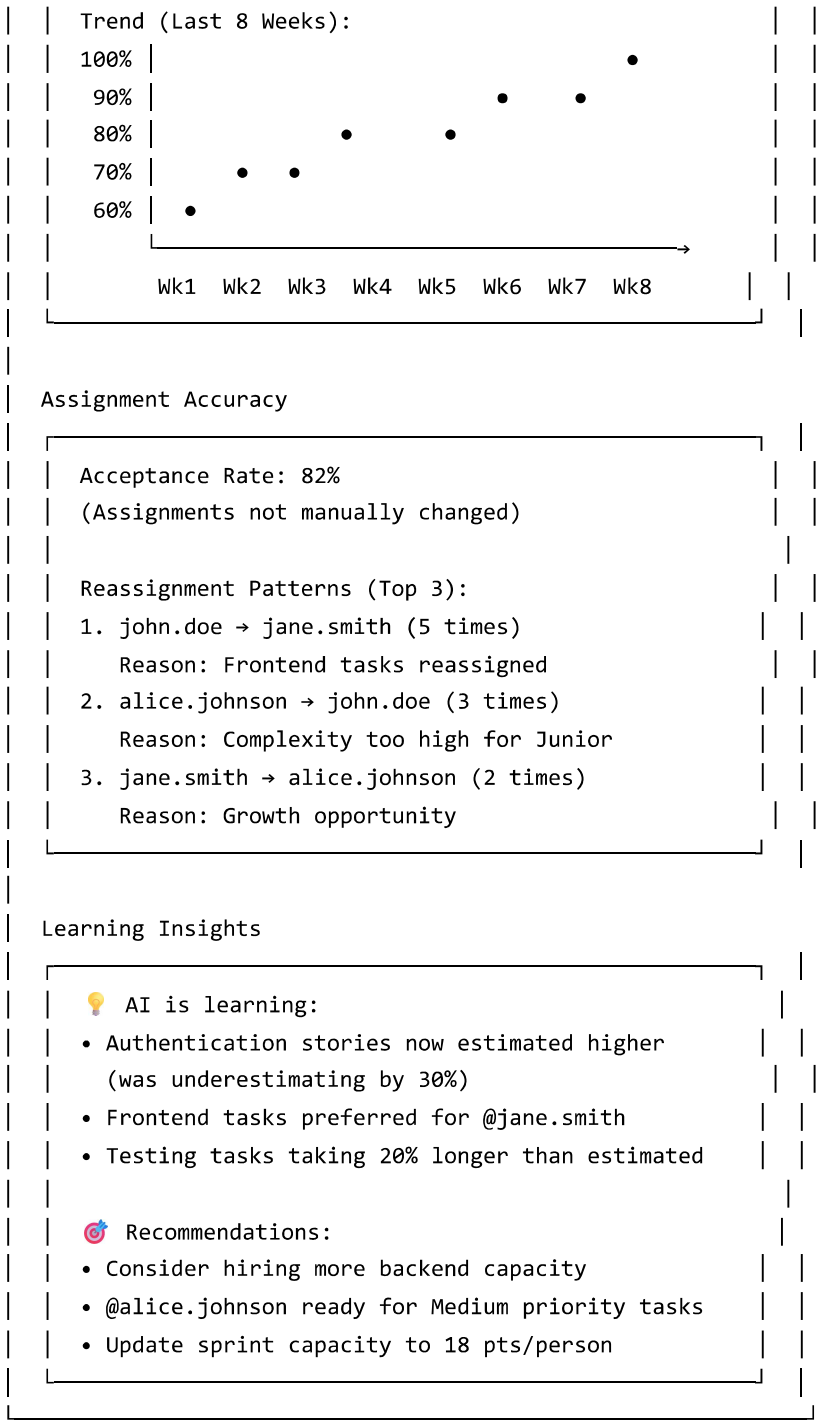
8.3 Team Capacity Dashboard





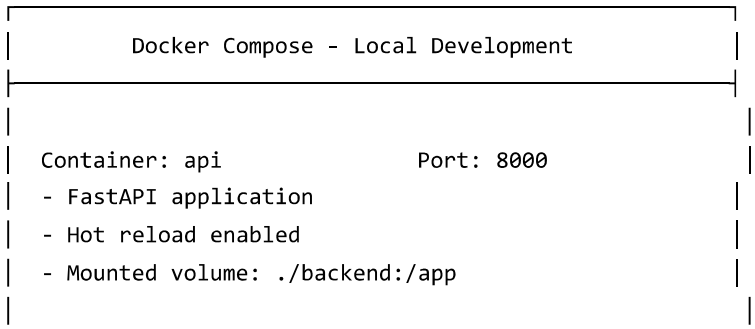
8.4 Analytics Dashboard





9. Deployment Architecture

9.1 Development Environment (Docker Compose)



```
│  Container: worker            (internal)              │
│  - Celery worker (4 concurrency)                      │
│  - Handles async tasks                                │
│                                                        │
│  Container: beat              (internal)              │
│  - Celery beat scheduler                              │
│  - Periodic tasks (capacity sync, queue processing)   │
│                                                        │
│  Container: flower            Port: 5555              │
│  - Celery monitoring UI                               │
│  - Real-time task tracking                            │
│                                                        │
│  Container: frontend          Port: 3000              │
│  - React development server                           │
│  - Hot module replacement                             │
│  - Mounted volume: ./frontend/src:/app/src            │
│                                                        │
│  Container: postgres          Port: 5432              │
│  - PostgreSQL 15                                      │
│  - Volume: postgres_data:/var/lib/postgresql/data     │
│  - Health check enabled                               │
│                                                        │
│  Container: redis             Port: 6379              │
│  - Redis 7 (cache + broker)                           │
│  - Volume: redis_data:/data                           │
│  - Persistence enabled                                │
│                                                        │
│  Container: chromadb          Port: 8001              │
│  - Vector database                                    │
│  - Volume: chroma_data:/chroma                        │
│                                                        │
│  Container: slack-bot         (internal)              │
│  - Slack integration (optional)                       │
│  - Socket mode (no public URL)                        │
│                                                        │
│  Network: jira-ai-network (bridge)                    │
│  Volumes: postgres_data, redis_data, chroma_data      │
└────────────────────────────────────────────────────────┘
```

13. API Endpoints

POST /api/prompt/create-story
Description: Create Jira story from natural language prompt
Request Body:

```
  {
    "prompt": "string (required, min 10 chars)",
    "issue_type": "Story|Task|Bug (default: Story)",
    "priority": "Highest|High|Medium|Low (default: Medium)",
    "project_key": "string (optional)",
    "epic_key": "string (optional)",
    "labels": ["string"] (optional),
```

```
    "auto_breakdown": boolean (default: true),
    "auto_estimate": boolean (default: true),
    "auto_assign": boolean (default: true)
  }
Response: 200 OK
  {
    "request_id": "uuid",
    "status": "processing|completed|failed",
    "generated_story": {
      "title": "string",
      "description": "string",
      "acceptance_criteria": ["string"],
      "estimated_points": integer
    },
    "jira_issue_key": "string (if completed)",
    "jira_url": "string (if completed)",
    "created_at": "datetime"
  }

---

GET /api/prompt/story-status/{request_id}
Description: Get status of story creation request
Parameters:
  - request_id: UUID (path parameter)
Response: 200 OK
  {
    "request_id": "uuid",
    "status": "pending|processing|completed|failed",
    "generated_story": {...},
    "jira_issue_key": "string",
    "error_message": "string (if failed)"
  }

---

POST /api/prompt/chat
Description: Conversational interface for story creation
Request Body:
  {
    "message": "string",
    "session_id": "string (optional)",
    "context": {} (optional)
  }
Response: 200 OK
  {
    "response": "string",
    "suggestions": ["string"],
    "actions": [
      {
        "type": "create_story",
```

```
          "data": {...}
        }
      ],
      "session_id": "string"
    }
```

```
POST /api/prompt/suggest-estimation
Description: Get estimation suggestion for existing story
Request Body:
  {
    "story_title": "string",
    "story_description": "string"
  }
Response: 200 OK
  {
    "estimated_points": integer,
    "reasoning": "string",
    "confidence": float,
    "similar_stories": [...]
  }

13.2 Capacity Management
GET /api/capacity/team
Description: Get team capacity overview
Response: 200 OK
  {
    "total_team_capacity": integer,
    "total_used_capacity": integer,
    "available_capacity": integer,
    "utilization_percentage": float,
    "team_size": integer,
    "available_members": integer,
    "members_by_status": {
      "available": ["username"],
      "busy": ["username"],
      "overloaded": ["username"]
    }
  }
```

```
GET /api/capacity/member/{username}
Description: Get individual member capacity
Parameters:
  - username: string (path parameter)
Response: 200 OK
  {
    "username": "string",
    "display_name": "string",
```

```
    "seniority": "string",
    "current_story_points": integer,
    "max_story_points": integer,
    "available_story_points": integer,
    "current_ticket_count": integer,
    "availability_percentage": float,
    "availability_status": "available|busy|overloaded|ooo",
    "is_out_of_office": boolean,
    "skills": ["string"],
    "preferred_work": ["string"]
  }

---

POST /api/capacity/mark-ooo
Description: Mark team member as out of office
Request Body:
  {
    "username": "string",
    "start_date": "datetime",
    "end_date": "datetime",
    "reason": "string",
    "partial_capacity": float (optional, 0-100)
  }
Response: 200 OK
  {
    "status": "success",
    "message": "string"
  }

13.3 Assignment

POST /api/assignment/assign-ticket
Description: Manually trigger ticket assignment
Request Body:
  {
    "issue_key": "string",
    "priority": "string",
    "estimated_points": integer,
    "required_skills": ["string"]
  }
Response: 200 OK
  {
    "assigned_to": "string",
    "display_name": "string",
    "assignment_score": float,
    "reasoning": "string",
    "alternatives": [
      {
        "username": "string",
        "score": float
```

```
        }
      ]
    }
```

```
GET /api/assignment/queue
Description: Get current assignment queue
Response: 200 OK
  {
    "queued_count": integer,
    "items": [
      {
        "issue_key": "string",
        "priority": "string",
        "estimated_points": integer,
        "attempts": integer,
        "reason": "string"
      }
    ]
  }
```

13.4 Analytics

```
GET /api/analytics/estimation-accuracy
Description: Get estimation accuracy metrics
Response: 200 OK
  {
    "acceptance_rate": float,
    "average_error": float,
    "total_estimations": integer,
    "monthly_trend": [
      {
        "month": "string",
        "error": float,
        "count": integer
      }
    ]
  }
```

```
GET /api/analytics/assignment-accuracy
Description: Get assignment accuracy metrics
Response: 200 OK
  {
    "acceptance_rate": float,
    "total_assignments": integer,
    "reassignments": integer,
    "common_reassignment_patterns": [
      {
```

```
          "from": "string",
          "to": "string",
          "count": integer
        }
      ]
  }
```

13.5 Webhooks

POST /api/webhook/jira

Description: Jira webhook handler (internal)

Request Body: Jira webhook payload

Response: 200 OK

```
  {
    "status": "received|processed|ignored"
  }
```