Resume parsing Java Microservice & Python resume parser Microservice

Contents:

6. **Final DB Persistence Logic**
    - Full-replace strategy
    - Transactional writes
    - Deletes Redis draft after success
    - Idempotent submits

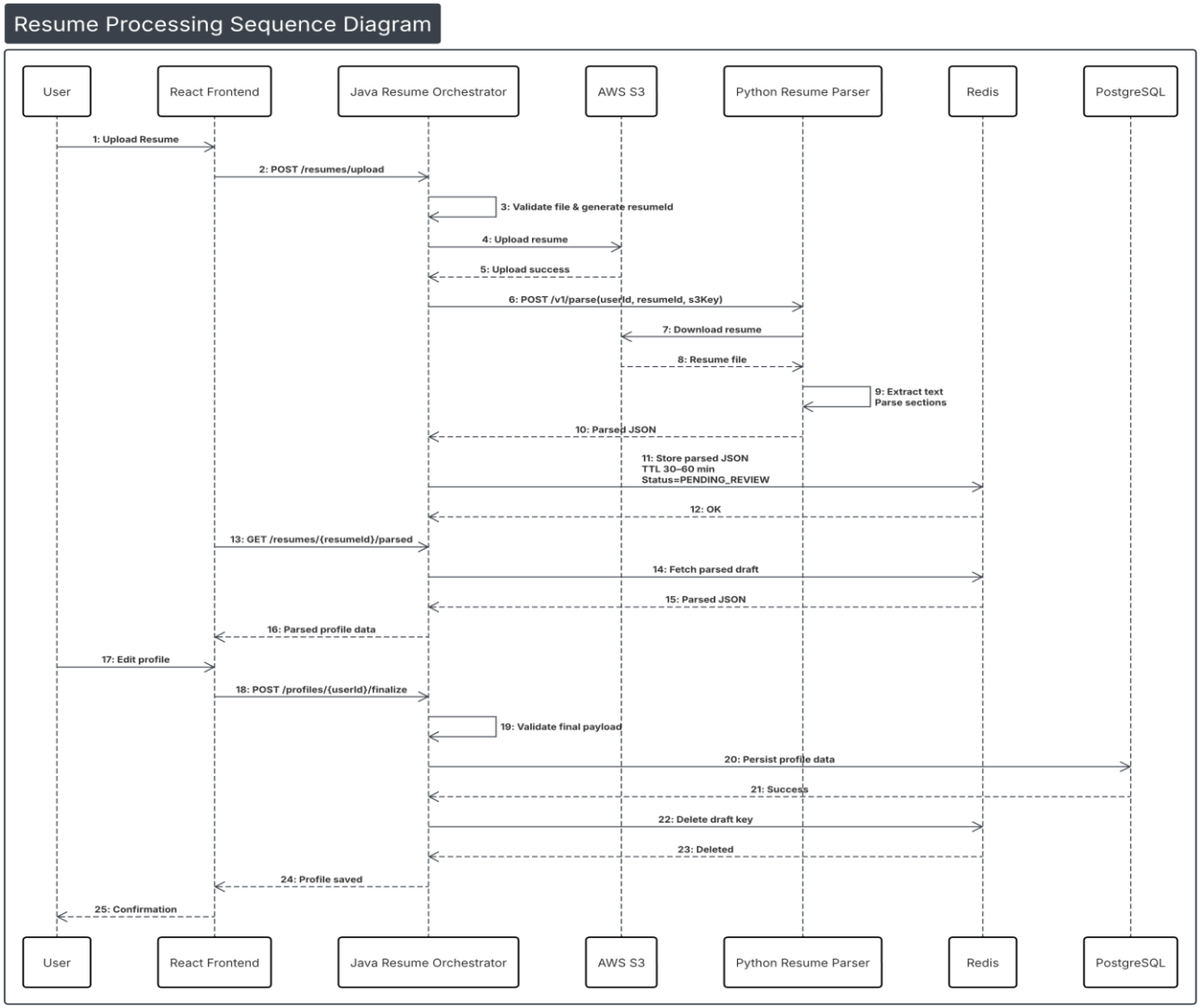
7. **Error & Edge-Case Handling**
    - Parser failure fallback
    - Partial data handling
    - Redis expiry recovery
    - DB rollback logic

8. **Performance & Scaling Responsibilities**
    - Stateless design
    - Separate scaling from Python
    - Rate-limit uploads & parse triggers

9. **Security Boundaries**
    - Auth validation
    - Owns access to Redis & DB
    - Python service treated as internal dependency

10. **QA / Acceptance Criteria (Java-owned)**
    - No DB writes before submit
    - No data loss
    - Redis TTL sliding verified
    - Multiple work experiences handled

Sequence Diagram:



Overall Orchestration Role

1) Components

- **React UI**
  - Upload resume
  - Review parsed profile
  - Edit + submit final profile
- **Java Resume Orchestrator Service**
  - Owns workflow + validations
  - Calls Python parser
  - Stores temporary state in Redis
  - Persists final user-confirmed profile into DB
- **Python Resume Parser Service**
  - Parse-only (S3 → JSON)
  - No DB, no Redis
- **AWS S3**
  - Stores raw resumes
- **Redis**
  - Stores "parsed draft profile" for user review/edit (TTL)
- **PostgreSQL**
  - Stores finalized user profile per your schema

2) APIs (clean contract)

A) React → Java

1. POST /resumes/upload
   Input: multipart file
   Output: {resumeId, status}
2. GET /resumes/{resumeId}/parsed
   Output: parsed JSON from Redis
3. POST /profiles/{userId}/finalize
   Input: final edited profile JSON
   Output: {status: SAVED}

B) Java → Python

4. POST /v1/parse
   Input: {userId,resumeId,s3Bucket,s3Key,fileType}
   Output: {personal,experience,education,skills,meta}

3) Data contracts

Parsed draft (stored in Redis)

Key:

- resume:parsed:{userId}:{resumeId}

Value:

- Parsed JSON + metadata:
  - createdAt
  - expiresAt
  - parseStatus (PENDING_REVIEW, FAILED, PARTIAL)

TTL:

- 30–60 minutes - **(Auto scales based on the user activity)**

Final submit (sent to Java, persisted to DB)

- Same shape as UI sections
- No parser metadata
- Java maps to DB tables

4) End-to-end workflow (sequence)

1. **React uploads resume** → Java
2. Java validates + uploads to **S3**
3. Java calls **Python parse**
4. Python downloads from S3 → extracts → sections → returns JSON
5. Java stores parsed JSON in **Redis (TTL)**
   Status: PENDING_REVIEW
6. React calls GET parsed → renders editable UI
7. User edits → submits final profile
8. Java validates + writes to **Postgres**
9. Java deletes Redis key

5) State machine (workflow status)

ResumeDraftStatus

- UPLOADED
- PARSING
- PENDING_REVIEW
- PARTIAL_READY
- FAILED
- SUBMITTED
- PERSISTED

6) Failure handling rules

- Python parse failure:
  - Java retries 2 times
  - If still fails → store {parseStatus: FAILED} in Redis and still let user manually fill UI
- Partial extraction:
  - Store PARTIAL_READY
  - UI shows empty fields ready for user completion
- Redis expiry:
  - React gets “expired” → user re-uploads or re-triggers parse

7) Security & scaling (minimum)

- Python service:
  - Internal network only
  - S3 access via IAM role
- Java service:
  - Auth required (JWT/session)
  - Redis in private subnet
- Rate limits:
  - Per user resume upload to avoid abuse

2. API Contracts (React ↔ Java)

A. React → Java

Upload Resume
POST /resumes/upload
multipart/form-data
file=<resume>
Response
{ "resumeId": "uuid", "status": "UPLOADED" }

Fetch Parsed Draft

GET /resumes/{resumeId}/parsed
Response

```
{
  "personal": {},
  "experience": [],
  "education": [],
  "skills": [],
  "meta": {
    "sectionsFound": ["experience","education"],
    "partial": true
  }
}
```

Finalize Profile

POST /profiles/{userId}/finalize

```
{
  "personal": {},
  "experience": [],
  "education": [],
```

```
  "skills": []
}
```

Response

```
{ "status": "PERSISTED" }
```

B. Java → Python

Parse Resume

POST /v1/parse

```
{
  "userId": "uuid",
  "resumeId": "uuid",
  "s3Bucket": "bucket",
  "s3Key": "path/to/file",
  "fileType": "PDF|DOCX|TXT|JPG"
}
```

Response

```
{
  "personal": {},
  "experience": [],
  "education": [],
  "skills": [],
  "meta": {
    "sectionsFound": [],
    "partial": false
  }
}
```

C. Redis (Java-owned)

Key

resume:parsed:{userId}:{resumeId}

Value

{

```
  "parsedDraft": { ... },
  "status": "PENDING_REVIEW|PARTIAL_READY|FAILED",
  "createdAt": "iso",
  "expiresAt": "iso"
}
```

TTL: 30–60 minutes – Auto Scales based on the user activity.

D. Errors (uniform)

```
{ "code": "PARSER_FAILED|DRAFT_EXPIRED|VALIDATION_ERROR", "message": "..." }
```

3. Integration with Python Parser

A. Parsing (Python) – JSON sample

- The **experience section** is split into **multiple experience blocks**
- Each block becomes **one WorkExperience object**

Example (Python output):

```
"experience": [
  {
    "jobTitle": "Software Engineer",
    "company": "ABC Corp",
    "startDate": "2021",
    "endDate": "2023",
    "description": "...",
    "skills": ["Java", "Spring"]
  },
  {
    "jobTitle": "Intern",
    "company": "XYZ Ltd",
    "startDate": "2019",
```

```
    "endDate": "2020",
    "description": "...",
    "skills": ["Python"]
  }
]
```

Work Experience Sample JSON:

```
{
  "experience": [
    {
      "jobTitle": "Software Engineer",
      "company": "ABC Corp"
    },
    {
      "jobTitle": "Intern",
      "company": "XYZ Ltd"
    }
  ]
}
```

B. Temporary Storage (Redis)

- The **entire array** of experiences is stored
- Order preserved (top-to-bottom from resume)

Redis value:

```
{
  "experience": [ {...}, {...}, {...} ]
}
```

C. Frontend (React)

- Work Experience rendered as a **repeatable section**
- User can:

    - Edit each experience
    - Add new experience
    - Delete existing experience
    - Reorder experiences

Each experience has a **frontend-generated ID** (UUID) for stable editing.

D. Final Submission (React → Java)

- Entire **experience array** is submitted
- Java does **full replace** for work experiences:
    - Deletes existing rows
    - Inserts submitted list (transactional)

E. DB Persistence (Java → PostgreSQL)

- One row per experience in work_experiences
- One-to-many relationship:
- user_profile 1 → N work_experiences
- Skills linked via join table:
- work_experience → work_experience_skills

F. Why this is correct

- Resumes are inherently **multi-experience**
- UI flexibility preserved
- Simple persistence logic
- No partial merges
- Easy future extensions (ordering, tagging)

4. Redis Draft Management

A. Purpose

Redis is used to store temporary parsed resume drafts for:

- User review
- User editing
- Session continuity

Redis is NOT a source of truth and MUST NOT be used for long-term storage.

B. Key Design

resume:parsed:{userId}:{resumeId}

Properties

- Namespaced
- User-scoped
- Resume-scoped
- Collision-safe

C. Value Structure

```
{
  "parsedDraft": {
    "personal": {},
    "experience": [],
    "education": [],
    "skills": []
  },
  "status": "PENDING_REVIEW | PARTIAL_READY | FAILED",
  "createdAt": "ISO-8601",
  "lastAccessedAt": "ISO-8601"
}
```

Notes

- No confidence flags (future enhancement)
- No DB-related metadata
- Only draft data required for UI

D. TTL Strategy (CRITICAL)

❗ TTL is SLIDING, not fixed

- Redis TTL MUST be refreshed on user activity
- Draft expires only after inactivity

Default TTL

- 60 minutes (sliding)

Example:

EXPIRE resume:parsed:{userId}:{resumeId} 3600

E. What counts as "User Activity"

Any of the following MUST refresh TTL:

- Editing a field
- Adding / deleting / reordering sections
- Auto-save event
- Explicit keep-alive ping
- Any interaction on Profile Edit page

F. Required Backend Contract

Keep-alive Endpoint

POST /resumes/{resumeId}/keep-alive

Backend responsibility

- Validate session
- Refresh Redis TTL
- Update lastAccessedAt

G. Lifecycle Rules

Creation

- Created after successful Python parsing
- Status set to PENDING_REVIEW or PARTIAL_READY

Active Editing

- TTL refreshed continuously via sliding expiration
- Draft remains available indefinitely while user is active

Inactivity

- If no activity for TTL duration → draft expires automatically

Final Submission

- After user submits final profile:
  - Draft is deleted immediately
  - DB becomes source of truth

H. Redis Expiry Handling

If Redis key is missing:

1. Check resume exists in S3
2. Re-trigger Python parsing
3. Create new Redis draft
4. Continue UI flow

❌ No resume re-upload required

I. Access Rules

Component	Redis Access
React	❌
Python	❌
Java	☑ (exclusive owner)

J. Failure States

Scenario	Redis Behavior
Python parse fails	Store empty/partial draft, status = FAILED
Partial extraction	status = PARTIAL_READY
User abandons	TTL expiry
Session timeout	TTL naturally expires

K. Non-Functional Requirement (MANDATORY)

Redis draft TTL must follow a sliding expiration model.
Draft data must not expire while the user is actively editing the profile, regardless of total elapsed time.
Draft expiration must occur only after inactivity.

L. Why this design is correct

- Prevents data loss
- Aligns with session management
- Scales horizontally
- Clean separation of concerns
- Industry-standard (ATS-grade)

5. React State Model (Profile Editing)

React State Model – FINAL

A. Core principle

- Single source of truth = React state
- Redis is just a backend cache
- DB is written only on final submit

B. Top-level state shape

```
ProfileDraft {
  personal: PersonalDetails
  experience: WorkExperience[]
  education: Education[]
  skills: string[]
  meta: {
    resumeId: string
    lastSyncedAt: string
  }
}
```

C. Work Experience (repeatable)

```
WorkExperience {
  id: string       // frontend UUID
  jobTitle: string
```

company: string
startDate?: string
endDate?: string
description?: string
skills: string[]
}

- Supports multiple experiences
- Add / edit / delete / reorder
- Order preserved via array index

D. Editing rules

- Any edit updates local React state
- No DB writes
- Optional auto-save → Java → Redis
- Keep-alive ping sent on edits

E. Page lifecycle

On load

1. Fetch parsed draft from Java
2. Initialize state

While editing

- Update local state
- Send keep-alive every 3–5 min

On submit

1. Validate required fields
2. Strip all metadata
3. POST final payload to Java

F. What React should NOT do

- ❌ Call Python directly
- ❌ Access Redis
- ❌ Persist partial data to DB

- ❌ Rely on parser correctness

G. Failure handling (UI)

- Redis expired → Java re-parses → UI reloads
- Parser partial → UI shows empty editable sections
- Validation error → highlight fields

H. Why this model works

- Simple mental model
- Supports long edits
- No data loss
- Scales with complexity

6. DB Persistence Architecture

A. Trigger

- Only on final submit
- Source: React → Java
- Payload: fully user-confirmed profile

B. Persistence strategy

- Full replace (not merge)
- Single transaction
- Either everything commits or nothing

C. Transaction flow

1. Validate payload
2. Begin transaction
3. Delete existing profile data (scoped to user)
4. Insert new data
5. Commit
6. Delete Redis draft

Rollback on any failure.

D. Table-level mapping (high level)

Personal Details

- 1 row per user

Work Experiences

- N rows per user
- Preserve order using display_order

Work Experience Skills

- N rows per experience

Education

- N rows per user

User Skills

- Deduplicated global skill list

E. Idempotency rule

- Same payload submitted twice → same DB state
- No side effects
- Safe retries

F. Validation rules (Java)

- Required fields present
- Date consistency
- Duplicate experience handling
- Skill normalization (basic)

G. Failure handling

- Validation fails → reject, no DB write
- DB error → rollback, Redis draft retained
- User can retry submit safely

H. Why this works

- Simple logic
- No stale data
- Handles multiple experiences cleanly

- Easy to extend later

7. Performance & Scaling

A. Overall design principle

- All services are stateless
- Scale horizontally at every layer
- No shared in-memory state

B. Component-wise scaling

React

- Static build (CDN / CloudFront)
- Scales automatically
- No bottlenecks

Java Resume Orchestrator

- Stateless Spring Boot service
- Scale via:
    - ECS/EKS auto-scaling
    - CPU / request-based scaling
- Redis + DB externalized → safe scaling

Python Resume Parser

- Stateless
- CPU-bound (OCR / PDF parsing)
- Scale via:
    - Horizontal pods
    - Separate autoscaling policy from Java
- Can be throttled independently

Redis

- Single logical store
- Use managed Redis (ElastiCache)
- Configure:

    - Memory eviction (LRU)
    - Max memory alerts
- Sliding TTL prevents runaway growth

PostgreSQL

- Writes only on final submit
- Read-heavy operations avoided
- Scale via:
    - Vertical scaling initially
    - Read replicas later (if needed)

C. Traffic patterns (important)

Operation	Frequency
Resume upload	Low
Parsing	Medium
Profile editing	High
DB writes	Low

👉 Parsing and DB writes are decoupled, preventing bottlenecks.

D. Rate limiting & protection

- Resume upload: per-user limit
- Parsing trigger: debounced
- Prevents abuse & cost spikes

E. Retry & backoff

- Python parse: retry max 2 times
- Exponential backoff
- Prevents parser overload

F. Long-running edit sessions

- Redis sliding TTL
- Keep-alive pings
- Supports hour-long edits safely

G. Observability (minimum)

- Request latency
- Parse duration
- Redis hit/miss ratio
- DB commit success rate

H. Why this scales well

- No synchronous chains
- No DB in hot path
- Clear separation of concerns
- Each layer scales independently

8. QA / Test Strategy

8.1. Test levels

A. Unit Tests
Who: Individual service owners
What:

- Python:
  - Text extraction (PDF/DOCX/JPG)
  - Section splitting
  - Multiple work experience parsing
- Java:
  - Payload validation
  - Redis TTL refresh logic
  - DB mapping logic
- React:
  - State updates
  - Add/edit/delete experience

B. Integration Tests
What to validate end-to-end service interaction
Flows

- Java → Python parse success
- Java → Python parse failure + retry
- Java → Redis write/read
- Redis sliding TTL refresh
- Java → DB final submit

C. End-to-End (E2E) Tests
Simulate real user journeys
Scenarios:

- Upload → parse → edit → submit
- Partial parse → manual completion → submit
- Redis expiry → auto re-parse → continue
- Session timeout → re-login → re-parse
- Multiple work experiences handling

8.2 Critical edge cases (must test)

Case	Expected
Empty resume	UI editable
Image-only resume	OCR output
Parser failure	Manual entry allowed
Redis TTL expiry	Auto re-parse
Long edit (>1 hr)	No data loss
Multiple experiences	All preserved
Duplicate submit	Idempotent

8.3 Redis-specific QA (VERY IMPORTANT)

- TTL extends on:
  - Field edit
  - Auto-save
  - Keep-alive ping
- TTL expires only after inactivity
- Draft deleted after final submit

8.4 Performance testing

- Concurrent resume uploads
- Concurrent parse requests
- Redis memory growth under load
- Parser CPU utilization
- DB write throughput

8.5 Security testing

- Unauthorized access blocked
- S3 access restricted
- Redis not exposed externally
- Payload validation enforced

8.6 Regression testing

Must re-run when:

- Parser logic changes
- UI fields change
- DB schema evolves
- Redis TTL logic modified

8.7 Acceptance criteria (LOCKED)

- ❌ No user data loss
- ❌ No forced re-upload
- ☑ Multiple experiences handled
- ☑ Manual override always allowed
- ☑ Draft survives long edits
- ☑ DB written only on submit