ChatGPT

FinFlow Platform Architecture

FinFlow is an AI-powered financial operations platform built with a **microservices** architecture. Each domain (authentication, payments, accounting, AI analytics, credit scoring) is implemented as an independent service with its own database and API. Services communicate via **REST** APIs (defined with OpenAPI 3.0) and **Kafka** event streams for asynchronous workflows. This approach ensures decoupling and scalability: microservices can be developed, deployed, and scaled independently [1] [2]. For example, when an invoice is created in the Accounting service, it publishes an `invoice_created` event to Kafka, which the Payments service (or AI modules) can consume. Kafka acts as a highly available message broker that enables resilient, event-driven coordination across services [2].

- **Decentralized Data:** Each service manages its own Postgres database (using Prisma or SQLAlchemy), avoiding a single shared schema [3].
- **Independent Deployment:** Services are isolated "black boxes" that expose APIs but hide their internal logic [4].
- **Polyglot Stack:** We use TypeScript/Node.js for most services, Python/FastAPI for AI modules, and React for the frontend, choosing the best tools for each domain [5].
- **DevOps ("You build it, you run it"):** Teams own individual services end-to-end, following CI/CD pipelines and container orchestration [6].

This architecture supports FinFlow's core domains (identity, payments, accounting, analytics) with strong security and compliance (PCI-DSS, GDPR) built into each layer [7] [8]. All APIs are documented with OpenAPI 3.0 (the industry standard for RESTful APIs [9]), and the code includes robust validation, logging, and error handling. Below we detail each component.

Frontend (React + TypeScript)

The frontend is a React app written in TypeScript, using **React Router** for navigation and **Redux/Context** for global state (e.g. user session, invoices). We use **Tailwind CSS** for responsive styling. Key UI features include:

- **Dashboard:** Shows KPIs (revenue, expenses, cash flow forecasts) with charts. We might use a library like Recharts or Chart.js.
- **Invoice UI:** Forms for creating/editing invoices, with client-side validation and loading states.
- **Cash Flow Analytics:** Tables and graphs showing historical cash flow and forecasts.
- **Payment Tracking:** Interfaces to view payment statuses and trigger new payments.
- **Responsive Design:** Tailwind classes ensure the app works on desktop and mobile.

A typical React structure:

```
finflow-frontend/
├── public/
├── src/
```

```
│   ├── components/        # Reusable UI components (charts, form inputs)
│   ├── pages/             # Page-level components (Dashboard, Invoices, 
Payments)
│   ├── store/             # Redux slices or Context providers
│   ├── App.tsx            # Main router setup
│   └── index.tsx          # Application entrypoint
└── package.json
```

For example, `src/App.tsx` sets up routes using React Router v6:

```
import { BrowserRouter as Router, Routes, Route } from 'react-router-dom';
import Dashboard from './pages/Dashboard';
import Invoices from './pages/Invoices';
import Payments from './pages/Payments';
import Login from './pages/Login';

function App() {
  return (
    <Router>
      <Routes>
        <Route path="/login" element={<Login />} />
        <Route path="/dashboard" element={<Dashboard />} />
        <Route path="/invoices" element={<Invoices />} />
        <Route path="/payments" element={<Payments />} />
        {/* more routes */}
      </Routes>
    </Router>
  );
}

export default App;
```

Each page uses React hooks or Redux selectors to fetch data from backend APIs. State management via Redux (or Context) handles authentication tokens and global state. For example, a Redux slice might store `auth.user` and `auth.token` from the Auth service. All API calls include the JWT in headers (Bearer token).

Example invoice creation form (`InvoiceForm.tsx`):

```
import { useState } from 'react';
import axios from 'axios';

export default function InvoiceForm() {
  const [client, setClient] = useState('');
  const [amount, setAmount] = useState(0);
```

```
    const [dueDate, setDueDate] = useState('');
    const [loading, setLoading] = useState(false);
    const [error, setError] = useState('');

    const submitInvoice = async () => {
      setLoading(true);
      try {
        await axios.post('/api/accounting/invoices', { client, amount, dueDate });
        // On success, maybe show a toast
      } catch (err) {
        setError('Failed to create invoice');
      }
      setLoading(false);
    };

    return (
      <form onSubmit={(e) => { e.preventDefault(); submitInvoice(); }}>
        <input value={client} onChange={e => setClient(e.target.value)}
  placeholder="Client name" className="input" />
        <input type="number" value={amount} onChange={e =>
  setAmount(+e.target.value)} placeholder="Amount" className="input" />
        <input type="date" value={dueDate} onChange={e =>
  setDueDate(e.target.value)} className="input" />
        <button type="submit" disabled={loading} className="btn">
          {loading ? 'Creating...' : 'Create Invoice'}
        </button>
        {error && <p className="text-red-500">{error}</p>}
      </form>
    );
  }
```

Utility classes from Tailwind make this responsive. All API base URLs and feature toggles go into a `.env` file (e.g. `VITE_API_URL`). We also add unit tests for components and e2e tests (e.g. with Jest and React Testing Library) to ensure the UI works correctly.

Backend Microservices

Each backend service is a standalone containerized application, written in Node.js/TypeScript (Express or NestJS) or Python (FastAPI for AI). They all expose **OpenAPI 3.0**-compliant REST endpoints. All services connect to a **PostgreSQL** database (using an ORM like Prisma or Sequelize), and use environment variables (`.env`) for config.

Below is an overview of each service:

Authentication Service (Node.js + Express)

This service handles user registration, login, and OAuth2.0 flows, issuing JWTs for API authentication. It also manages roles and permissions.

- **Endpoints:** `POST /auth/register`, `POST /auth/login`, `GET /auth/me`, OAuth callbacks (`/auth/oauth2/redirect`), etc.
- **Tech:** Express.js, Passport.js (for OAuth2), bcrypt for password hashing, jsonwebtoken for JWT.
- **DB Schema (Prisma):** Users table with fields `(id, email, hashedPassword, role, createdAt)`.
- **Security:** Passwords hashed; JWTs signed with a secret (`JWT_SECRET` from `.env`); refresh tokens stored in DB.
- **Example Code Snippet:**

```
// src/routes/auth.ts
import express from 'express';
import jwt from 'jsonwebtoken';
import bcrypt from 'bcrypt';
import { PrismaClient } from '@prisma/client';
const prisma = new PrismaClient();
const router = express.Router();

// Register endpoint
router.post('/register', async (req, res) => {
  const { email, password } = req.body;
  const hash = await bcrypt.hash(password, 10);
  const user = await prisma.user.create({ data: { email, hashedPassword: hash,
role: 'USER' } });
  res.json({ id: user.id, email: user.email });
});

// Login endpoint
router.post('/login', async (req, res) => {
  const { email, password } = req.body;
  const user = await prisma.user.findUnique({ where: { email } });
  if (!user || !await bcrypt.compare(password, user.hashedPassword)) {
    return res.status(401).json({ error: 'Invalid credentials' });
  }
  const token = jwt.sign({ sub: user.id, role: user.role },
process.env.JWT_SECRET!, { expiresIn: '1h' });
  res.json({ accessToken: token });
});

export default router;
```

- **OpenAPI Spec:** We include Swagger docs. For example, `/auth/login` might be documented as:

```
paths:
  /auth/login:
    post:
      summary: "User login"
      requestBody:
        content:
          application/json:
            schema:
              type: object
              properties:
                email: { type: string }
                password: { type: string }
      responses:
        '200':
          description: "JWT token"
          content:
            application/json:
              schema:
                type: object
                properties:
                  accessToken: { type: string }
```

All other routes are similarly defined. The Auth service's README details setup (OAuth client IDs, JWT secret, DB migrations) and includes its own `Dockerfile` and `.env.example` (showing `PORT`, `DATABASE_URL`, `JWT_SECRET`, OAuth client credentials, etc).

Security best practices are followed: JWTs for stateless auth, HTTPS enforced, and CORS configured tightly. We grant *roles* (e.g. `ADMIN`, `USER`) and apply middleware to protect certain routes (e.g. only `ADMIN` can view all users). For example, a middleware might check the JWT and `role` claim on each request.

Payments Service (Node.js + Express)

The Payments service handles initiating and tracking payments (credit card, ACH, etc.). It integrates with payment processors (e.g. Stripe or Plaid) via API/webhooks.

- **Endpoints:** `POST /payments/charge`, `GET /payments/:id`, `POST /payments/webhook` to receive external payment status updates.
- **DB Schema:** Payments table with `(id, userId, amount, currency, status, processorData, createdAt)`.
- **Integration:** Uses a payment SDK. Example: Stripe's Node library to create charges.
- **Example Code Snippet (charging):**

```
// src/routes/payments.ts
import express from 'express';
import stripe from 'stripe';
```

```
const stripeClient = new stripe(process.env.STRIPE_SECRET!, { apiVersion:
'2020-08-27' });
const router = express.Router();

router.post('/charge', async (req, res) => {
  const { amount, currency, source, userId } = req.body;
  try {
    const charge = await stripeClient.charges.create({ amount, currency,
source });
    // Save payment to DB (skipped for brevity)
    // Publish event to Kafka
    await kafkaProducer.send({ topic: 'payment_completed', messages: [{
key: charge.id, value: JSON.stringify({ userId, amount, status:
'completed' }) }] });
    res.json({ id: charge.id, status: charge.status });
  } catch (err) {
    console.error(err);
    res.status(500).json({ error: 'Payment failed' });
  }
});

// Webhook endpoint for async updates
router.post('/webhook', express.raw({ type: 'application/json' }), (req,
res) => {
  const sig = req.headers['stripe-signature'];
  const event = stripeClient.webhooks.constructEvent(req.body, sig!,
process.env.STRIPE_WEBHOOK_SECRET!);
  if (event.type === 'charge.succeeded') {
    const data = event.data.object as any;
    // Update DB and possibly re-publish to Kafka
  }
  res.sendStatus(200);
});
export default router;
```

- **Kafka Integration:** After a successful charge, the service publishes a `payment_completed` event (JSON) on a Kafka topic so accounting or analytics can update records. This decouples confirmation from processing.

- **PCI-DSS Compliance:** We never store raw card data; Stripe's tokens are used instead. Sensitive fields (like last4 digits) are encrypted before DB insertion. All traffic is over TLS.
- **Logging & Errors:** Uses a logger (e.g. Winston) to record payment attempts and errors.

Accounting Service (Node.js + Express)

This service manages invoices, ledgers, and reconciliations. It tracks money owed and paid.

- **Endpoints:** `POST /accounting/invoices` (create invoice), `GET /accounting/invoices`, `POST /accounting/ledger-entry`, etc.
- **DB Schema:** Invoices `(id, userId, client, amount, dueDate, status)`, JournalEntries `(id, invoiceId, debitAccount, creditAccount, amount, date)`.
- **Business Logic:** Creating an invoice generates initial ledger entries (debit AccountsReceivable, credit SalesRevenue). When a payment_completed event is consumed, the invoice status is updated to "Paid" and reversal ledger entry is created.
- **Example Code (invoice creation):**

```
router.post('/invoices', async (req, res) => {
  const { userId, client, amount, dueDate } = req.body;
  const invoice = await prisma.invoice.create({ data: { userId, client,
amount, dueDate, status: 'PENDING' } });
  // Publish invoice_created event
  await kafkaProducer.send({ topic: 'invoice_created', messages: [{ key:
invoice.id, value: JSON.stringify(invoice) }] });
  res.json(invoice);
});
```

- **Kafka Consumer:** Listens to `payment_completed` topic. On receiving a payment event, it marks the corresponding invoice as paid and records the bank transaction in its ledger.
- **OpenAPI:** Swagger docs define schemas for invoices and ledger entries. For example, `POST /accounting/ledger-entry` might accept `{ invoiceId, debitAccount, creditAccount, amount, date }`.
- **Validation:** We validate inputs (e.g. positive amounts, valid dates) and return clear error responses. Integration tests (e.g. using Jest + Supertest) ensure the endpoints behave as expected.
- **Reconciliation:** The service can reconcile ledger entries with payments, marking mismatches. This might be an asynchronous job or API action.

AI Analytics Service (Python + FastAPI)

This microservice provides machine learning analytics: real-time cash flow forecasting and automated expense categorization. It's built in Python (FastAPI) and uses libraries like **pandas**, **scikit-learn**, and **statsmodels**.

- **Endpoints:**
- `POST /analytics/forecast` takes recent revenue/expense time-series data and returns future cash flow predictions.
- `POST /analytics/categorize` takes a transaction description and returns a spending category (via a trained NLP model).
- **Modeling:** We train time-series models (e.g. ARIMA or Facebook Prophet) on historical data to forecast cash flow. We also train a text classification model (e.g. logistic regression or a small transformer) on labeled transactions.

- **Example Training Script:** (`train_models.py`)

```
import pandas as pd
from fbprophet import Prophet
from sklearn.linear_model import LogisticRegression
import joblib

# Synthetic data generation (for demo)
df = pd.read_csv('synthetic_transactions.csv')  # contains date, amount,
description, category
# Train forecast model
ts = df.groupby('date').sum().reset_index().rename(columns={'date':'ds',
'amount':'y'})
model = Prophet(yearly_seasonality=True)
model.fit(ts)
model.save('cashflow_prophet_model.pkl')
# Train categorization model
X = df['description']
y = df['category']
from sklearn.feature_extraction.text import TfidfVectorizer
vect = TfidfVectorizer()
X_vec = vect.fit_transform(X)
clf = LogisticRegression()
clf.fit(X_vec, y)
joblib.dump({'model': clf, 'vectorizer': vect}, 'category_model.joblib')
```

- **Inference Endpoints:** FastAPI loads these models at startup. For example:

```
from fastapi import FastAPI
import joblib
import pandas as pd

app = FastAPI()
forecast_model = Prophet().load('cashflow_prophet_model.pkl')
cat_data = joblib.load('category_model.joblib')
clf = cat_data['model']; vect = cat_data['vectorizer']

@app.post("/forecast")
def forecast(cashflow: dict):
    """
    Expects {'historical': [{'ds': '2024-01-01', 'y': 1000}, ...]}
    """
    df = pd.DataFrame(cashflow['historical'])
    future = forecast_model.make_future_dataframe(periods=30, freq='D')
    forecast = forecast_model.predict(future)
    return {"forecast":
```

```
forecast[['ds','yhat']].tail(30).to_dict(orient='records')}

@app.post("/categorize")
def categorize(text: str):
    vec = vect.transform([text])
    category = clf.predict(vec)[0]
    return {"category": category}
```

- **Kafka (Optional):** This service could consume events like `invoice_created` or `payment_completed` to trigger re-training or update predictions. For real-time forecasts, it might publish results to a `cashflow_forecast` topic.
- **Security:** Endpoints require a valid JWT (FastAPI's OAuth2PasswordBearer can verify tokens). We ensure that only authorized users can request analytics for their own data.
- **Example Data:** A `synthetic_transactions.csv` generator script creates fake transaction history for demo and tests. Unit tests verify model accuracy on known inputs.

Credit Engine (Python + FastAPI)

The Credit Engine provides AI-driven credit scoring and loan offers.

- **Endpoints:** `POST /credit/score` accepts applicant financial data (income, cash flow, credit history metrics) and returns a credit score or risk category. `GET /credit/offers` returns loan offers based on score.
- **Model:** We train a classification/regression model (e.g. using scikit-learn) on synthetic applicant data. This may factor in cash flow volatility, payment history (from Accounting data), and external credit API data.
- **Example Training:** (`train_credit_model.py`)

```
import numpy as np
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import train_test_split
from sklearn.datasets import make_classification
import joblib

# Generate synthetic features: income, numInvoices, avgCashflow,
paymentDelinquencies
X, y = make_classification(n_samples=1000, n_features=4, n_classes=3)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
clf = RandomForestRegressor()
clf.fit(X_train, y_train)
joblib.dump(clf, 'credit_score_model.pkl')
```

- **Inference:** Load model in FastAPI.

```
app = FastAPI()
credit_model = joblib.load('credit_score_model.pkl')
```

```
@app.post("/score")
def score(input: dict):
    """
    Expects {'income': float, 'numInvoices': int, 'avgCashflow': float,
'delinquencies': int}
    """
    features = [input['income'], input['numInvoices'],
input['avgCashflow'], input['delinquencies']]
    score = credit_model.predict([features])[0]
    return {"credit_score": score}
```

- **Loan Offers:** Based on the score, the service calculates interest rates or loan amounts. For example, if `score > 0.8`, offer up to $X loan at Y% APR.
- **Security & Compliance:** This service stores no personal data. All inputs are anonymized financial metrics. Responses include no sensitive personal info.

Event Streaming (Kafka)

Services communicate via Kafka topics for decoupled workflows. Key design points:

- **Topics & Schemas:** Each event type has a defined JSON schema (or Avro). E.g.:

```
// invoice_created event schema (JSON)
{
  "type": "object",
  "properties": {
    "id": { "type": "string" },
    "userId": { "type": "string" },
    "client": { "type": "string" },
    "amount": { "type": "number" },
    "dueDate": { "type": "string", "format": "date" },
    "status": { "type": "string" }
  },
  "required": ["id","userId","amount","dueDate","status"]
}
```

- **Producers:** Code examples above show using `kafkaProducer.send({ topic: ..., messages: [...] })`. In Node we might use kafkajs; in Python use kafka-python or confluent-kafka.
- **Consumers:** Services that need events set up consumers. For example, Accounting service might have:

```
import { Kafka } from 'kafkajs';
const kafka = new Kafka({ brokers: ['kafka:9092'] });
const consumer = kafka.consumer({ groupId: 'accounting-group' });
```

```
await consumer.connect();
await consumer.subscribe({ topic: 'payment_completed', fromBeginning:
false });
await consumer.run({
  eachMessage: async ({ message }) => {
    const evt = JSON.parse(message.value.toString());
    // Update invoice/payment status in DB
  }
});
```

- **Event-driven Benefits:** As noted, Kafka decouples producers/consumers and increases resilience [2]. For example, if the Analytics service is down, events queue in Kafka and replay when ready. The system can also rewind topics to recalc forecasts.

Security & Compliance

Security and compliance are baked in:

- **Authentication:** All REST endpoints require a valid JWT (OAuth2 Bearer) except auth routes. Middleware checks tokens against the Auth service's public key or secret.
- **Authorization:** Role-based access control (RBAC) ensures users can only access their data. For instance, a user can only query invoices they created (via `userId` checks in each query).
- **Encryption:** We use TLS (HTTPS) for all service communication. Sensitive fields (credit card last4, SSNs) are encrypted at rest (using database encryption features or application-level crypto).
- **PCI-DSS:** For payments, we follow PCI guidelines: no storage of full card data, use secure tokens, rotate encryption keys, and log all access attempts.
- **GDPR:** We implement data export/deletion endpoints. According to GDPR Article 17, individuals have the right to erasure [8]. Our services allow a user to request all their personal data and to delete it ("right to be forgotten"). For example, hitting `DELETE /auth/user/:id` would cascade and remove user-related data in other services (through Kafka events like `user_deleted`).
- **Audit Logging:** Every service logs sensitive actions (logins, payments, data exports) with timestamps and user IDs. Logs can be shipped to an ELK stack (Elastic) or Loki for auditing.

Overall, FinFlow follows OWASP and fintech best practices (least privilege, input validation, logging).

DevOps & Infrastructure

The platform is containerized and deployable via Docker Compose (for local) and Kubernetes (for production). Key points:

- **Dockerfiles:** Every service has a `Dockerfile`. Example (Node service):

```
FROM node:18-alpine
WORKDIR /app
COPY package*.json ./
RUN npm install --production
```

```
COPY . .
CMD ["node", "dist/index.js"]
```

And for Python:

```
FROM python:3.11-slim
WORKDIR /app
COPY requirements.txt ./
RUN pip install -r requirements.txt
COPY . .
CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "8000"]
```

- **docker-compose.yml:** Defines services (auth, payments, accounting, analytics, credit, kafka, zookeeper, postgres instances). It sets environment variables from `.env` files and links networks. This allows spinning up the entire stack locally.
- **Kubernetes/Helm:** We provide a `k8s/` folder with manifests. For each service we have a Deployment and Service. For example:

```
apiVersion: apps/v1
kind: Deployment
metadata: { name: auth-service }
spec:
  replicas: 3
  selector: { matchLabels: { app: auth } }
  template:
    metadata: { labels: { app: auth } }
    spec:
      containers:
      - name: auth
        image: registry/finflow-auth:latest
        ports: [{ containerPort: 3000 }]
        envFrom:
          - secretRef: { name: auth-secrets }
          - configMapRef: { name: auth-config }
---
apiVersion: v1
kind: Service
metadata: { name: auth-service }
spec:
  type: ClusterIP
  selector: { app: auth }
  ports: [{ port: 80, targetPort: 3000 }]
```

We also define a Kafka StatefulSet and Headless Service, a Postgres Deployment/Service for each DB, and a Redis (for caching sessions). Ingress (or a load balancer) fronts the API and UI.

- **CI/CD (GitHub Actions):** Our `.github/workflows/ci.yml` runs on PRs and pushes:
- Lint and unit tests for each service (e.g. `npm test`, `pytest`).
- Build Docker images and push to registry.
- Deploy to Kubernetes (using `kubectl` or Helm) on the `main` branch.
- Slack/Email notifications on failures. Example snippet:

```
name: CI
on: [push, pull_request]
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v3
    - uses: actions/setup-node@v3
      with: { 'node-version': '18' }
    - name: Install dependencies
      run: npm ci
    - name: Run tests
      run: npm test
    - name: Build Docker
      run: docker build -t ghcr.io/yourorg/finflow-auth:latest .
    - name: Push Docker
      uses: docker/login-action@v2
      with: { registry: 'ghcr.io', username: '${{ github.actor }}',
password: '${{ secrets.GITHUB_TOKEN }}' }
    - run: docker push ghcr.io/yourorg/finflow-auth:latest
```

- **Monitoring & Logging:** Each service exposes Prometheus metrics (via libraries like `prom-client` or `prometheus-fastapi-instrumentator`). We deploy Prometheus to scrape `/metrics` endpoints. Grafana dashboards visualize KPIs and health metrics (CPU, request latency per service). Logs (in JSON format) are collected via a sidecar or exported to Loki/ELK, tagged by service for troubleshooting.

By combining container orchestration and CI/CD, FinFlow can roll out updates with zero downtime (blue-green or canary deployments), enforce infrastructure-as-code, and automatically recover from failures.

Documentation & Diagrams

- **Main README:** Describes system setup (prerequisites: Docker, Node, Python, Kafka). It explains how to start with `docker-compose up` and how to run tests. It also lists all services and their purpose.
- **Service READMEs:** Each microservice has a README (Auth, Payments, Accounting, Analytics, Credit). These include API docs (or links to Swagger UI), environment variable descriptions (`.env.example`), and any service-specific notes (e.g. OAuth config for Auth).
- **OpenAPI/Swagger:** We include an OpenAPI JSON/YAML for each service (e.g. `auth-api.yaml`). Swagger UI or ReDoc can serve these for interactive API docs.

- **Architecture Diagram:** We provide a high-level diagram showing frontend, backend services, Kafka, and databases. Another diagram details event flows: e.g. flow from invoice creation to payment completion to accounting update.
- **Data Flow:** A sequence diagram or flowchart shows interactions: user login -> view dashboard -> create invoice -> service publishes event -> payments service handles charge -> accounting updates ledger, etc.

All code is commented for clarity. For example, complex logic in the Credit Engine notes why certain financial ratios are used. We follow SOLID principles in code: e.g. the Payments service separates controller/routes from payment provider adapters.

Each code module includes unit tests (e.g. Jest for Node, pytest for Python). We also have integration tests: e.g. using Docker to test that the frontend can log in against the Auth service, or that a Kafka event triggers the expected consumer behavior.

In summary, the FinFlow repository contains a top-level `README.md` and folders for: - `frontend/` (React app) - `services/auth/`, `services/payments/`, `services/accounting/` (Node services) - `services/analytics/`, `services/credit/` (Python services) - `k8s/` (Kubernetes manifests) - `.github/workflows/` (CI/CD pipelines)

This structure keeps components decoupled yet interoperable. With Docker and Kubernetes configuration included, the platform is deployment-ready. The design follows current best practices for fintech systems: event-driven microservices, secure auth, and comprehensive test/monitoring pipelines [1] [2] [9].

---

[1] [3] [4] [5] [6] How to Build a Fintech Platform with Microservices Architecture in 2025
https://www.velmie.com/post/how-to-build-a-fintech-platform-with-microservices

[2] Reference Architecture: Event-Driven Microservices with Apache Kafka | Heroku Dev Center
https://devcenter.heroku.com/articles/event-driven-microservices-with-apache-kafka

[7] Guide to Fintech Architecture with Challenges and Examples | DashDevs
https://dashdevs.com/blog/fintech-architecture/

[8] Art. 17 GDPR – Right to erasure ('right to be forgotten')
https://gdpr-info.eu/art-17-gdpr/

[9] OpenAPI 3.0: How to Design and Document APIs with the Latest OpenAPI Specification [Recorded Webinar]
https://swagger.io/blog/api-design/openapi-3-0-specification-training/