Teacher Timetable Extraction – Architectural Design Plan

Introduction
The aim of this solution is to design a serverless AWS architecture that allows teachers to upload PDF or image timetables and automatically extract structured time-block data for display. The system must handle variable document layouts and produce reliable results using OCR and large language models (LLM), with error handling and flexibility for future enhancements.

1. End-to-End Workflow
The processing pipeline follows these stages:

• **File upload** – The teacher uploads a timetable file (PDF or image) via a web UI. The frontend uses a secure REST API to send the file to an Amazon API Gateway endpoint; the upload Lambda stores the document in an Amazon S3 bucket, generates an upload ID and triggers a Step Functions workflow.

• **Pre-processing and OCR** – A Lambda function validates the file type/size and splits multi-page documents if necessary. It then invokes Amazon Textract to extract text, key-value pairs and tables from the document. Textract supports scanned PDFs and images and returns structured JSON containing table cells and their relationships. The raw Textract output is stored back in S3.

• **Post-processing & cleansing** – Another Lambda function normalises the text and table data: it removes noise, unifies date/time formats and detects table headers. For ambiguous values it records placeholders and flags them for review.

• **LLM parsing** – The cleaned text and extracted tables are passed to an LLM via Amazon Bedrock. A carefully designed prompt instructs the model to parse the timetable into structured JSON objects representing timeblocks (teacher name, class, day, start time, end time, subject, room). Few-shot examples provide guidance to the model. The LLM returns structured JSON; confidence scores are calculated by comparing with rule-based heuristics. If the LLM fails, a fallback parser or manual review is triggered.

• **Storage & notification** – The final structured time-block objects are stored in an Amazon DynamoDB table. Upload metadata (status, timestamps) is stored in a separate table. When processing completes, the workflow publishes a message to an Amazon SNS topic, and the frontend polls an API or uses WebSockets to retrieve the results.

• **Frontend display** – A React/Next.js frontend queries the API to fetch the structured timetable. It renders an interactive calendar view and allows teachers to correct any misparsed entries. The static assets are served via Amazon CloudFront.

The high-level flow can be depicted as:

```
Teacher UI → API Gateway → S3 (upload)
      → Step Functions state machine
        → Preprocess (Lambda)
        → Textract OCR
        → Post-process (Lambda)
        → LLM via Bedrock
        → Save to DynamoDB
        → Notify completion
```

Frontend fetches results from DynamoDB via API and displays timetable.
```

2. Recommended Tools, Frameworks & Languages

**File ingestion & preprocessing:**

- *Amazon S3* to store uploaded documents and intermediate outputs. S3 offers durable, scalable storage and integrates with event notifications.
- *Amazon API Gateway* to expose secure REST/HTTP APIs for uploads and retrieval. It provides authentication and throttling.
- *AWS Lambda* (Python) for serverless compute. Python provides strong libraries for PDF/image handling (pdf2image, Pillow) and integration with AWS SDKs.
- *AWS Step Functions* to orchestrate the workflow. The visual state machine simplifies sequencing, branching, error handling and retries.
- For multi-part uploads or large files, use *AWS S3 multipart upload* and *Signed URLs* on the frontend.

**OCR & document parsing:**

- *Amazon Textract* for extracting text, key-value pairs and tables from scanned documents. It supports multi-page PDFs and images and returns structured JSON.
- Optional fallback: open-source Tesseract OCR or libraries like Camelot/Tabula for specific PDF tables.

**LLM integration:**

- *Amazon Bedrock* for accessing foundation models (e.g., Claude, Llama). Bedrock offers managed inference endpoints, versioning and throughput control. Python SDKs (boto3 or LangChain) are used to call the model.
- Prompt engineering: design prompts with clear instructions and examples; set low temperature (0.0-0.2) for deterministic output; include JSON schema definitions.
- Use *LangChain* or *PromptLayer* to manage prompts and logging.

**Backend orchestration:**

- *AWS Step Functions* orchestrates asynchronous tasks and implements retries and catch blocks. It interacts with Lambda, Textract, Bedrock, DynamoDB and SNS.
- *Amazon SQS* as a decoupling mechanism and dead-letter queue for failed events.
- *API Gateway* + *Lambda* for REST endpoints; or use *AppSync* if GraphQL is preferred.

**Frontend rendering:**

- *React* with TypeScript and a UI framework (e.g., Material-UI) for the teacher portal.
- *AWS Amplify* or *Next.js* for hosting and authentication integration (Cognito).
- *Amazon CloudFront* CDN for caching static assets and ensuring low-latency access.

3. Suggested Database Schema for Timeblocks

Use DynamoDB because it is fully managed, serverless and automatically scales. Two tables can separate data and metadata:
```

TimeBlocks table
Attribute (Primary Key in bold)	Type	Description
UploadID + **BlockID**	String + Number (composite key)	Partition key is the upload/document identifier; sort key is the sequential block number.
TeacherName	String	Normalised teacher name.
Class	String	Class or grade.
Day	String	Day of week or date.
StartTime	String	Time in ISO 8601 (HH:MM).
EndTime	String	Time in ISO 8601 (HH:MM).
Subject	String	Subject taught.
Room	String	Classroom or location.
RawJSON	Map	Original parsed JSON from the LLM for traceability.
Confidence	Number	Confidence score (0-1) based on heuristics/LLM output.
CreatedAt	String (ISO 8601)	Timestamp when entry was created.

UploadMetadata table
Attribute (Primary Key in bold)	Type	Description
UploadID	String	Unique identifier for each upload.
TeacherID	String	ID of the teacher/uploader.
Status	String	Processing status: `Uploaded`, `Processing`, `Completed`, `Error`.
UploadedAt	String	Timestamp of upload.
CompletedAt	String	Timestamp when processing finished.
Errors	List	List of error messages if any.

4. LLM Integration Strategy

- **Pipeline location:** The LLM is used after OCR and preliminary cleansing. Cleaned text and tables are concatenated and sent to the model to derive structured timeblocks.

- **Prompt strategy:** Use a system prompt that defines the role: “You are a timetable parser.” Provide explicit JSON schema definitions and few-shot examples of input and expected output. To ensure reproducibility, set the temperature parameter to a low value and restrict the model to respond only with valid JSON. Logging prompts and outputs in a secure store facilitates auditing.

- **Accuracy & reproducibility:** Combine LLM parsing with deterministic heuristics. Use Step Functions to retry the call upon transient errors. Validate the JSON against a schema; if invalid or low confidence (<0.8), route to a fallback path that either invokes a secondary model or flags for manual review. Keep prompts versioned and store them in a repository so that changes are controlled.

5. Error Handling & Fallbacks

- **Bad uploads:** Validate file type (PDF, PNG/JPG) and size at the API layer. Reject unsupported formats with meaningful error messages. Use S3 pre-signed upload URLs to prevent large payloads from hitting the API.

- **Ambiguous data/missing fields:** After OCR and LLM parsing, run rule-based checks (e.g., verify that start times precede end times, days are valid). If fields are missing or ambiguous, store them with null/placeholder values and flag them in the UI for teacher correction. Maintain an SQS dead-letter queue for failed processes to allow re-processing.

- **Service errors:** Step Functions includes `Retry` and `Catch` blocks to handle transient errors from Textract or Bedrock. If an error persists, the workflow transitions to a failure state and publishes a notification. Use CloudWatch alarms and DLQs for monitoring.

6. Flexibility & Future-proofing

- **Modular architecture:** Each stage (upload, OCR, LLM parsing, storage) is decoupled using serverless components and messaging. New services (e.g., alternative OCR/LLM providers) can be integrated by adding states in the Step Functions definition without redesigning the entire pipeline.

- **Configuration & versioning:** Store configuration parameters (model version, prompt templates) in AWS Systems Manager Parameter Store. Use environment variables to switch between models or adjust thresholds.

- **Scalability:** S3 and DynamoDB scale automatically. Lambda scales concurrently based on demand. Step Functions ensures orchestrated scaling. CloudFront caches content globally for a responsive UI.

- **Extensibility:** Additional features like timetable conflict detection, analytics, or integration with calendars can be implemented by reading from the DynamoDB data. The frontend can evolve separately because the API remains consistent.

This architectural design plan provides a robust, serverless foundation for extracting teachers' timetables from uploaded documents, leveraging managed AWS services for scalability and reliability.