Based on the detailed analysis, let me help implement the SCORM runtime tracking and metrics functionality. Let's focus on the three most critical components:

1. First, let's enhance the SCORM model to track detailed runtime data:

from django.db import models

from django.conf import settings

from datetime import timedelta

class SCORMTrackingData(models.Model):

user = models.ForeignKey(settings.AUTH\_USER\_MODEL, on\_delete=models.CASCADE)

scorm = models.ForeignKey('SCORM', on\_delete=models.CASCADE)

current\_lesson\_location = models.CharField(max\_length=255, blank=True)

total\_time = models.DurationField(default=timedelta)

session\_time = models.DurationField(default=timedelta)

score\_raw = models.DecimalField(max\_digits=5, decimal\_places=2, null=True)

score\_scaled = models.DecimalField(max\_digits=5, decimal\_places=2, null=True)

completion\_status = models.CharField(

max\_length=20,

choices=[

('not\_attempted', 'Not Attempted'),

('incomplete', 'Incomplete'),

('completed', 'Completed'),

('passed', 'Passed'),

('failed', 'Failed')

],

default='not\_attempted'

)

success\_status = models.CharField(

max\_length=20,

choices=[

('unknown', 'Unknown'),

('passed', 'Passed'),

('failed', 'Failed')

],

default='unknown'

)

entry = models.CharField(max\_length=20, default='ab-initio')

exit\_status = models.CharField(max\_length=20, blank=True)

progress\_measure = models.DecimalField(

max\_digits=5,

decimal\_places=2,

null=True,

help\_text="Learner's progress from 0 to 1"

)

created\_at = models.DateTimeField(auto\_now\_add=True)

updated\_at = models.DateTimeField(auto\_now=True)

class Meta:

unique\_together = ('user', 'scorm')

indexes = [

models.Index(fields=['user', 'scorm']),

models.Index(fields=['completion\_status']),

models.Index(fields=['success\_status'])

]

def \_\_str\_\_(self):

return f"{self.user.username} - {self.scorm.title}"

2. Next, let's create a robust view for handling SCORM runtime updates:

from django.http import JsonResponse

from django.views.decorators.csrf import csrf\_exempt

from django.contrib.auth.decorators import login\_required

from django.shortcuts import get\_object\_or\_404

from datetime import datetime, timedelta

import json

import logging

logger = logging.getLogger(\_\_name\_\_)

def parse\_scorm\_time(time\_str):

"""Parse SCORM time string (e.g., 'PT1H30M45S') into timedelta"""

try:

time\_str = time\_str.upper().replace('PT','')

hours = minutes = seconds = 0

if 'H' in time\_str:

h\_split = time\_str.split('H')

hours = float(h\_split[0])

time\_str = h\_split[1]

if 'M' in time\_str:

m\_split = time\_str.split('M')

minutes = float(m\_split[0])

time\_str = m\_split[1]

if 'S' in time\_str:

seconds = float(time\_str.replace('S',''))

return timedelta(hours=hours, minutes=minutes, seconds=seconds)

except Exception as e:

logger.error(f"Error parsing SCORM time: {str(e)}")

return timedelta(0)

@login\_required

@csrf\_exempt

def scorm\_runtime\_update(request, scorm\_id):

"""Handle SCORM runtime data updates"""

if request.method != "POST":

return JsonResponse({"error": "Method not allowed"}, status=405)

try:

scorm = get\_object\_or\_404(SCORM, id=scorm\_id)

data = json.loads(request.body)

# Get or create tracking record

tracking\_data, created = SCORMTrackingData.objects.get\_or\_create(

user=request.user,

scorm=scorm

)

# Update tracking data based on SCORM 1.2 CMI fields

if 'cmi.core.lesson\_location' in data:

tracking\_data.current\_lesson\_location = data['cmi.core.lesson\_location']

if 'cmi.core.score.raw' in data:

raw\_score = float(data['cmi.core.score.raw'])

tracking\_data.score\_raw = raw\_score

tracking\_data.score\_scaled = raw\_score / 100.0

if 'cmi.core.session\_time' in data:

session\_time = parse\_scorm\_time(data['cmi.core.session\_time'])

tracking\_data.session\_time = session\_time

tracking\_data.total\_time += session\_time

if 'cmi.core.lesson\_status' in data:

status = data['cmi.core.lesson\_status'].lower()

if status in ['completed', 'incomplete', 'not attempted', 'failed', 'passed']:

tracking\_data.completion\_status = status

if status in ['passed', 'failed']:

tracking\_data.success\_status = status

if 'cmi.progress\_measure' in data:

tracking\_data.progress\_measure = float(data['cmi.progress\_measure'])

tracking\_data.save()

# Update aggregate metrics on SCORM package

update\_scorm\_metrics(scorm)

return JsonResponse({

"status": "success",

"tracking\_data": {

"completion\_status": tracking\_data.completion\_status,

"success\_status": tracking\_data.success\_status,

"score": tracking\_data.score\_scaled,

"total\_time": str(tracking\_data.total\_time),

"progress": tracking\_data.progress\_measure

}

})

except Exception as e:

logger.error(f"SCORM runtime update error: {str(e)}")

return JsonResponse({"error": str(e)}, status=500)

def update\_scorm\_metrics(scorm):

"""Update aggregate metrics for a SCORM package"""

try:

tracking\_records = SCORMTrackingData.objects.filter(scorm=scorm)

# Calculate metrics

total\_learners = tracking\_records.count()

completed\_count = tracking\_records.filter(

completion\_status\_\_in=['completed', 'passed']

).count()

avg\_score = tracking\_records.filter(

score\_scaled\_\_isnull=False

).aggregate(Avg('score\_scaled'))['score\_scaled\_\_avg'] or 0

avg\_time = tracking\_records.aggregate(

Avg('total\_time')

)['total\_time\_\_avg'] or timedelta(0)

# Update SCORM package metrics

scorm.runtime\_tracking.update({

'total\_learners': total\_learners,

'completion\_rate': (completed\_count / total\_learners \* 100) if total\_learners > 0 else 0,

'average\_score': avg\_score,

'average\_time': str(avg\_time),

'completed\_count': completed\_count,

'last\_updated': datetime.now().isoformat()

})

scorm.save()

except Exception as e:

logger.error(f"Error updating SCORM metrics: {str(e)}")

3. Finally, let's create a React component for displaying SCORM metrics in the dashboard:

import React, { useState, useEffect } from 'react';

import { Card, CardHeader, CardTitle, CardContent } from '@/components/ui/card';

import { LineChart, Line, XAxis, YAxis, CartesianGrid, Tooltip, Legend, ResponsiveContainer, BarChart, Bar } from 'recharts';

import { Users, Clock, Award, BookOpen } from 'lucide-react';

const SCORMMetricsDashboard = ({ scormId }) => {

const [metrics, setMetrics] = useState({

totalLearners: 0,

completionRate: 0,

averageScore: 0,

averageTime: '0:00:00',

completedCount: 0,

progressHistory: [],

scoreDistribution: []

});

useEffect(() => {

const fetchMetrics = async () => {

try {

const response = await fetch(`/api/scorm/${scormId}/metrics/`);

const data = await response.json();

setMetrics(data);

} catch (error) {

console.error('Error fetching SCORM metrics:', error);

}

};

fetchMetrics();

const interval = setInterval(fetchMetrics, 30000); // Refresh every 30 seconds

return () => clearInterval(interval);

}, [scormId]);

return (

<div className="space-y-6">

<div className="grid grid-cols-1 md:grid-cols-4 gap-4">

{/\* Total Learners \*/}

<Card>

<CardContent className="pt-4">

<div className="flex items-center space-x-4">

<Users className="h-8 w-8 text-blue-500" />

<div>

<p className="text-sm text-gray-500">Total Learners</p>

<p className="text-2xl font-bold">{metrics.totalLearners}</p>

</div>

</div>

</CardContent>

</Card>

{/\* Completion Rate \*/}

<Card>

<CardContent className="pt-4">

<div className="flex items-center space-x-4">

<BookOpen className="h-8 w-8 text-green-500" />

<div>

<p className="text-sm text-gray-500">Completion Rate</p>

<p className="text-2xl font-bold">{metrics.completionRate.toFixed(1)}%</p>

<p className="text-sm text-gray-500">{metrics.completedCount} completed</p>

</div>

</div>

</CardContent>

</Card>

{/\* Average Score \*/}

<Card>

<CardContent className="pt-4">

<div className="flex items-center space-x-4">

<Award className="h-8 w-8 text-purple-500" />

<div>

<p className="text-sm text-gray-500">Average Score</p>

<p className="text-2xl font-bold">{metrics.averageScore.toFixed(1)}%</p>

</div>

</div>

</CardContent>

</Card>

{/\* Average Time \*/}

<Card>

<CardContent className="pt-4">

<div className="flex items-center space-x-4">

<Clock className="h-8 w-8 text-orange-500" />

<div>

<p className="text-sm text-gray-500">Average Time</p>

<p className="text-2xl font-bold">{metrics.averageTime}</p>

</div>

</div>

</CardContent>

</Card>

</div>

{/\* Progress History Chart \*/}

<Card>

<CardHeader>

<CardTitle>Progress History</CardTitle>

</CardHeader>

<CardContent>

<div className="h-64">

<ResponsiveContainer width="100%" height="100%">

<LineChart data={metrics.progressHistory}>

<CartesianGrid strokeDasharray="3 3" />

<XAxis

dataKey="date"

tickFormatter={(date) => new Date(date).toLocaleDateString()}

/>

<YAxis />

<Tooltip />

<Legend />

<Line

type="monotone"

dataKey="progress"

stroke="#2563eb"

name="Progress (%)"

/>

<Line

type="monotone"

dataKey="score"

stroke="#16a34a"

name="Score (%)"

/>

</LineChart>

</ResponsiveContainer>

</div>

</CardContent>

</Card>

{/\* Score Distribution \*/}

<Card>

<CardHeader>

<CardTitle>Score Distribution</CardTitle>

</CardHeader>

<CardContent>

<div className="h-64">

<ResponsiveContainer width="100%" height="100%">

<BarChart data={metrics.scoreDistribution}>

<CartesianGrid strokeDasharray="3 3" />

<XAxis dataKey="range" />

<YAxis />

<Tooltip />

<Bar dataKey="count" fill="#4f46e5" name="Learners" />

</BarChart>

</ResponsiveContainer>

</div>

</CardContent>

</Card>

</div>

);

};

export default SCORMMetricsDashboard;

To implement these components:

1. Create the migration for the new SCORMTrackingData model:

python manage.py makemigrations courses

python manage.py migrate

2. Add the API endpoint in courses/urls.py:

path('api/scorm/<int:scorm\_id>/metrics/', views.get\_scorm\_metrics, name='scorm\_metrics'),

3. Update the instructor and learner dashboard templates to include the metrics component:

<script type="module">

import SCORMMetricsDashboard from './SCORMMetricsDashboard.js';

// Initialize for each SCORM package

{% for scorm in scorm\_content %}

new SCORMMetricsDashboard({

scormId: {{ scorm.id }},

container: document.getElementById('metrics-{{ scorm.id }}')

});

{% endfor %}

</script>

* + We already have basic SCORM models and views implemented
  + The folder structure for SCORM content is established
  + Dashboard integration is working

Let us look into a synthesized recommendation that combines the best of both approaches:

If we have to ensure your recommendations and scorm tracking need to be implemented review following:

**SCORM Runtime Tracker Component – as above**

### Enhanced SCORM Runtime View

# In courses/views.py

@login\_required

@csrf\_exempt

def scorm\_runtime\_update(request, course\_id):

if request.method == "POST":

try:

data = json.loads(request.body)

scorm\_package = get\_object\_or\_404(SCORM, course\_id=course\_id)

# Get or create learner progress record

progress, created = ScormLearnerProgress.objects.get\_or\_create(

scorm=scorm\_package,

learner=request.user

)

# Update progress data

if 'cmi.core.lesson\_location' in data:

progress.current\_position = data['cmi.core.lesson\_location']

if 'cmi.core.score.raw' in data:

progress.raw\_score = float(data['cmi.core.score.raw'])

progress.scaled\_score = float(data['cmi.core.score.raw']) / 100.0

if 'cmi.core.session\_time' in data:

session\_time = parse\_scorm\_time(data['cmi.core.session\_time'])

progress.session\_time = session\_time

progress.total\_time += session\_time

if 'cmi.core.lesson\_status' in data:

progress.completion\_status = data['cmi.core.lesson\_status']

# Store raw SCORM data

progress.progress\_data.update(data)

progress.save()

# Update aggregate stats on SCORM package

update\_scorm\_aggregate\_stats(scorm\_package)

return JsonResponse({

"status": "success",

"progress": {

"completion\_status": progress.completion\_status,

"score": progress.scaled\_score,

"total\_time": str(progress.total\_time)

}

})

except Exception as e:

logger.error(f"Runtime tracking error: {str(e)}")

return JsonResponse({"error": str(e)}, status=500)

return JsonResponse({"error": "Invalid method"}, status=400)

def parse\_scorm\_time(time\_str):

"""Parse SCORM time format (e.g., 'PT1H22M33S') into timedelta"""

try:

hours = minutes = seconds = 0

time\_str = time\_str.upper()

if 'H' in time\_str:

h\_split = time\_str.split('H')

hours = float(h\_split[0].replace('PT', ''))

time\_str = h\_split[1]

if 'M' in time\_str:

m\_split = time\_str.split('M')

minutes = float(m\_split[0])

time\_str = m\_split[1]

if 'S' in time\_str:

seconds = float(time\_str.replace('S', ''))

return timedelta(hours=hours, minutes=minutes, seconds=seconds)

except Exception as e:

logger.error(f"Error parsing SCORM time: {str(e)}")

return timedelta(0)

def update\_scorm\_aggregate\_stats(scorm\_package):

"""Update aggregate statistics for the SCORM package"""

try:

progress\_records = ScormLearnerProgress.objects.filter(scorm=scorm\_package)

# Calculate aggregates

completed\_count = progress\_records.filter(

completion\_status='completed'

).count()

total\_learners = progress\_records.count()

if total\_learners > 0:

completion\_rate = (completed\_count / total\_learners) \* 100

avg\_score = progress\_records.filter(

scaled\_score\_\_isnull=False

).aggregate(Avg('scaled\_score'))['scaled\_score\_\_avg'] or 0

# Update SCORM package stats

scorm\_package.runtime\_tracking.update({

'completion\_rate': completion\_rate,

'average\_score': avg\_score,

'total\_learners': total\_learners,

'completed\_learners': completed\_count

})

scorm\_package.save()

except Exception as e:

logger.error(f"Error updating SCORM stats: {str(e)}")

### SCORM Metrics Dashboard Component

import React, { useState, useEffect } from 'react';

import { Card, CardHeader, CardTitle, CardContent } from '@/components/ui/card';

import { LineChart, Line, XAxis, YAxis, CartesianGrid, Tooltip, Legend, ResponsiveContainer, BarChart, Bar } from 'recharts';

import { Book, Clock, Award, Users } from 'lucide-react';

const ScormMetricsDashboard = ({ courseId }) => {

const [metrics, setMetrics] = useState({

completionRate: 0,

averageScore: 0,

totalLearners: 0,

completedLearners: 0,

timeData: [],

scoreDistribution: []

});

useEffect(() => {

const fetchMetrics = async () => {

try {

const response = await fetch(`/api/courses/${courseId}/scorm-metrics/`);

const data = await response.json();

setMetrics(data);

} catch (error) {

console.error('Error fetching metrics:', error);

}

};

fetchMetrics();

const interval = setInterval(fetchMetrics, 60000); // Refresh every minute

return () => clearInterval(interval);

}, [courseId]);

return (

<div className="space-y-6">

{/\* Overview Cards \*/}

<div className="grid grid-cols-1 md:grid-cols-4 gap-4">

<Card>

<CardContent className="pt-4">

<div className="flex items-center space-x-4">

<Users className="h-8 w-8 text-blue-500" />

<div>

<p className="text-sm text-gray-500">Total Learners</p>

<p className="text-2xl font-bold">{metrics.totalLearners}</p>

</div>

</div>

</CardContent>

</Card>

<Card>

<CardContent className="pt-4">

<div className="flex items-center space-x-4">

<Book className="h-8 w-8 text-green-500" />

<div>

<p className="text-sm text-gray-500">Completion Rate</p>

<p className="text-2xl font-bold">{metrics.completionRate}%</p>

</div>

</div>

</CardContent>

</Card>

<Card>

<CardContent className="pt-4">

<div className="flex items-center space-x-4">

<Award className="h-8 w-8 text-purple-500" />

<div>

<p className="text-sm text-gray-500">Average Score</p>

<p className="text-2xl font-bold">{metrics.averageScore}%</p>

</div>

</div>

</CardContent>

</Card>

<Card>

<CardContent className="pt-4">

<div className="flex items-center space-x-4">

<Clock className="h-8 w-8 text-orange-500" />

<div>

<p className="text-sm text-gray-500">Completed</p>

<p className="text-2xl font-bold">{metrics.completedLearners}</p>

</div>

</div>

</CardContent>

</Card>

</div>

{/\* Time Spent Chart \*/}

<Card>

<CardHeader>

<CardTitle>Time Spent Trends</CardTitle>

</CardHeader>

<CardContent>

<div className="h-64">

<ResponsiveContainer width="100%" height="100%">

<LineChart data={metrics.timeData}>

<CartesianGrid strokeDasharray="3 3" />

<XAxis dataKey="date" />

<YAxis />

<Tooltip />

<Legend />

<Line type="monotone" dataKey="averageTime" stroke="#2563eb" name="Avg Time (min)" />

</LineChart>

</ResponsiveContainer>

</div>

</CardContent>

</Card>

{/\* Score Distribution \*/}

<Card>

<CardHeader>

<CardTitle>Score Distribution</CardTitle>

</CardHeader>

<CardContent>

<div className="h-64">

<ResponsiveContainer width="100%" height="100%">

<BarChart data={metrics.scoreDistribution}>

<CartesianGrid strokeDasharray="3 3" />

<XAxis dataKey="range" />

<YAxis />

<Tooltip />

<Bar dataKey="count" fill="#4f46e5" />

</BarChart>

</ResponsiveContainer>

</div>

</CardContent>

</Card>

</div>

);

};

export default ScormMetricsDashboard;

### Enhanced SCORM Models

from django.db import models

from django.conf import settings

from datetime import timedelta

class SCORMTrackingData(models.Model):

user = models.ForeignKey(settings.AUTH\_USER\_MODEL, on\_delete=models.CASCADE)

scorm = models.ForeignKey('SCORM', on\_delete=models.CASCADE)

current\_lesson\_location = models.CharField(max\_length=255, blank=True)

total\_time = models.DurationField(default=timedelta)

session\_time = models.DurationField(default=timedelta)

score\_raw = models.DecimalField(max\_digits=5, decimal\_places=2, null=True)

score\_scaled = models.DecimalField(max\_digits=5, decimal\_places=2, null=True)

completion\_status = models.CharField(

max\_length=20,

choices=[

('not\_attempted', 'Not Attempted'),

('incomplete', 'Incomplete'),

('completed', 'Completed'),

('passed', 'Passed'),

('failed', 'Failed')

],

default='not\_attempted'

)

success\_status = models.CharField(

max\_length=20,

choices=[

('unknown', 'Unknown'),

('passed', 'Passed'),

('failed', 'Failed')

],

default='unknown'

)

entry = models.CharField(max\_length=20, default='ab-initio')

exit\_status = models.CharField(max\_length=20, blank=True)

progress\_measure = models.DecimalField(

max\_digits=5,

decimal\_places=2,

null=True,

help\_text="Learner's progress from 0 to 1"

)

created\_at = models.DateTimeField(auto\_now\_add=True)

updated\_at = models.DateTimeField(auto\_now=True)

class Meta:

unique\_together = ('user', 'scorm')

indexes = [

models.Index(fields=['user', 'scorm']),

models.Index(fields=['completion\_status']),

models.Index(fields=['success\_status'])

]

def \_\_str\_\_(self):

return f"{self.user.username} - {self.scorm.title}"

### SCORM Runtime Views

from django.http import JsonResponse

from django.views.decorators.csrf import csrf\_exempt

from django.contrib.auth.decorators import login\_required

from django.shortcuts import get\_object\_or\_404

from datetime import datetime, timedelta

import json

import logging

logger = logging.getLogger(\_\_name\_\_)

def parse\_scorm\_time(time\_str):

"""Parse SCORM time string (e.g., 'PT1H30M45S') into timedelta"""

try:

time\_str = time\_str.upper().replace('PT','')

hours = minutes = seconds = 0

if 'H' in time\_str:

h\_split = time\_str.split('H')

hours = float(h\_split[0])

time\_str = h\_split[1]

if 'M' in time\_str:

m\_split = time\_str.split('M')

minutes = float(m\_split[0])

time\_str = m\_split[1]

if 'S' in time\_str:

seconds = float(time\_str.replace('S',''))

return timedelta(hours=hours, minutes=minutes, seconds=seconds)

except Exception as e:

logger.error(f"Error parsing SCORM time: {str(e)}")

return timedelta(0)

@login\_required

@csrf\_exempt

def scorm\_runtime\_update(request, scorm\_id):

"""Handle SCORM runtime data updates"""

if request.method != "POST":

return JsonResponse({"error": "Method not allowed"}, status=405)

try:

scorm = get\_object\_or\_404(SCORM, id=scorm\_id)

data = json.loads(request.body)

# Get or create tracking record

tracking\_data, created = SCORMTrackingData.objects.get\_or\_create(

user=request.user,

scorm=scorm

)

# Update tracking data based on SCORM 1.2 CMI fields

if 'cmi.core.lesson\_location' in data:

tracking\_data.current\_lesson\_location = data['cmi.core.lesson\_location']

if 'cmi.core.score.raw' in data:

raw\_score = float(data['cmi.core.score.raw'])

tracking\_data.score\_raw = raw\_score

tracking\_data.score\_scaled = raw\_score / 100.0

if 'cmi.core.session\_time' in data:

session\_time = parse\_scorm\_time(data['cmi.core.session\_time'])

tracking\_data.session\_time = session\_time

tracking\_data.total\_time += session\_time

if 'cmi.core.lesson\_status' in data:

status = data['cmi.core.lesson\_status'].lower()

if status in ['completed', 'incomplete', 'not attempted', 'failed', 'passed']:

tracking\_data.completion\_status = status

if status in ['passed', 'failed']:

tracking\_data.success\_status = status

if 'cmi.progress\_measure' in data:

tracking\_data.progress\_measure = float(data['cmi.progress\_measure'])

tracking\_data.save()

# Update aggregate metrics on SCORM package

update\_scorm\_metrics(scorm)

return JsonResponse({

"status": "success",

"tracking\_data": {

"completion\_status": tracking\_data.completion\_status,

"success\_status": tracking\_data.success\_status,

"score": tracking\_data.score\_scaled,

"total\_time": str(tracking\_data.total\_time),

"progress": tracking\_data.progress\_measure

}

})

except Exception as e:

logger.error(f"SCORM runtime update error: {str(e)}")

return JsonResponse({"error": str(e)}, status=500)

def update\_scorm\_metrics(scorm):

"""Update aggregate metrics for a SCORM package"""

try:

tracking\_records = SCORMTrackingData.objects.filter(scorm=scorm)

# Calculate metrics

total\_learners = tracking\_records.count()

completed\_count = tracking\_records.filter(

completion\_status\_\_in=['completed', 'passed']

).count()

avg\_score = tracking\_records.filter(

score\_scaled\_\_isnull=False

).aggregate(Avg('score\_scaled'))['score\_scaled\_\_avg'] or 0

avg\_time = tracking\_records.aggregate(

Avg('total\_time')

)['total\_time\_\_avg'] or timedelta(0)

# Update SCORM package metrics

scorm.runtime\_tracking.update({

'total\_learners': total\_learners,

'completion\_rate': (completed\_count / total\_learners \* 100) if total\_learners > 0 else 0,

'average\_score': avg\_score,

'average\_time': str(avg\_time),

'completed\_count': completed\_count,

'last\_updated': datetime.now().isoformat()

})

scorm.save()

except Exception as e:

logger.error(f"Error updating SCORM metrics: {str(e)}")

### SCORM Metrics Dashboard

import React, { useState, useEffect } from 'react';

import { Card, CardHeader, CardTitle, CardContent } from '@/components/ui/card';

import { LineChart, Line, XAxis, YAxis, CartesianGrid, Tooltip, Legend, ResponsiveContainer, BarChart, Bar } from 'recharts';

import { Users, Clock, Award, BookOpen } from 'lucide-react';

const SCORMMetricsDashboard = ({ scormId }) => {

const [metrics, setMetrics] = useState({

totalLearners: 0,

completionRate: 0,

averageScore: 0,

averageTime: '0:00:00',

completedCount: 0,

progressHistory: [],

scoreDistribution: []

});

useEffect(() => {

const fetchMetrics = async () => {

try {

const response = await fetch(`/api/scorm/${scormId}/metrics/`);

const data = await response.json();

setMetrics(data);

} catch (error) {

console.error('Error fetching SCORM metrics:', error);

}

};

fetchMetrics();

const interval = setInterval(fetchMetrics, 30000); // Refresh every 30 seconds

return () => clearInterval(interval);

}, [scormId]);

return (

<div className="space-y-6">

<div className="grid grid-cols-1 md:grid-cols-4 gap-4">

{/\* Total Learners \*/}

<Card>

<CardContent className="pt-4">

<div className="flex items-center space-x-4">

<Users className="h-8 w-8 text-blue-500" />

<div>

<p className="text-sm text-gray-500">Total Learners</p>

<p className="text-2xl font-bold">{metrics.totalLearners}</p>

</div>

</div>

</CardContent>

</Card>

{/\* Completion Rate \*/}

<Card>

<CardContent className="pt-4">

<div className="flex items-center space-x-4">

<BookOpen className="h-8 w-8 text-green-500" />

<div>

<p className="text-sm text-gray-500">Completion Rate</p>

<p className="text-2xl font-bold">{metrics.completionRate.toFixed(1)}%</p>

<p className="text-sm text-gray-500">{metrics.completedCount} completed</p>

</div>

</div>

</CardContent>

</Card>

{/\* Average Score \*/}

<Card>

<CardContent className="pt-4">

<div className="flex items-center space-x-4">

<Award className="h-8 w-8 text-purple-500" />

<div>

<p className="text-sm text-gray-500">Average Score</p>

<p className="text-2xl font-bold">{metrics.averageScore.toFixed(1)}%</p>

</div>

</div>

</CardContent>

</Card>

{/\* Average Time \*/}

<Card>

<CardContent className="pt-4">

<div className="flex items-center space-x-4">

<Clock className="h-8 w-8 text-orange-500" />

<div>

<p className="text-sm text-gray-500">Average Time</p>

<p className="text-2xl font-bold">{metrics.averageTime}</p>

</div>

</div>

</CardContent>

</Card>

</div>

{/\* Progress History Chart \*/}

<Card>

<CardHeader>

<CardTitle>Progress History</CardTitle>

</CardHeader>

<CardContent>

<div className="h-64">

<ResponsiveContainer width="100%" height="100%">

<LineChart data={metrics.progressHistory}>

<CartesianGrid strokeDasharray="3 3" />

<XAxis

dataKey="date"

tickFormatter={(date) => new Date(date).toLocaleDateString()}

/>

<YAxis />

<Tooltip />

<Legend />

<Line

type="monotone"

dataKey="progress"

stroke="#2563eb"

name="Progress (%)"

/>

<Line

type="monotone"

dataKey="score"

stroke="#16a34a"

name="Score (%)"

/>

</LineChart>

</ResponsiveContainer>

</div>

</CardContent>

</Card>

{/\* Score Distribution \*/}

<Card>

<CardHeader>

<CardTitle>Score Distribution</CardTitle>

</CardHeader>

<CardContent>

<div className="h-64">

<ResponsiveContainer width="100%" height="100%">

<BarChart data={metrics.scoreDistribution}>

<CartesianGrid strokeDasharray="3 3" />

<XAxis dataKey="range" />

<YAxis />

<Tooltip />

<Bar dataKey="count" fill="#4f46e5" name="Learners" />

</BarChart>

</ResponsiveContainer>

</div>

</CardContent>

</Card>

</div>

);

};

export default SCORMMetricsDashboard;

The integrated solution I'm recommending combines both approaches by:

1. **Enhanced Structure**:
   * Created a SCORMHandler class to encapsulate SCORM-related logic
   * Maintains your existing folder structure and upload workflow
   * Adds robust runtime tracking
2. **Improved Upload Process**:
   * Validates SCORM package structure
   * Handles extraction to the correct directory
   * Updates database with proper launch\_url
3. **Runtime Tracking**:
   * Captures detailed SCORM runtime data
   * Updates aggregate metrics
   * Maintains historical data for analytics
4. **Key Improvements Over Current Implementation**:
   * Better error handling and logging
   * More structured code organization
   * Enhanced metrics tracking
   * Maintains backward compatibility

For implementation, you would need to:

1. Create migrations for any new models:

bash

Copy

python manage.py makemigrations

python manage.py migrate

1. Update your URLs:

python

Copy

urlpatterns = [

path('scorm/upload/', views.upload\_scorm, name='upload\_scorm'),

path('scorm/runtime/<int:scorm\_id>/update/', views.scorm\_runtime\_update, name='scorm\_runtime\_update'),

path('scorm/play/<int:course\_id>/', views.play\_scorm, name='play\_scorm'),

]

1. Update your templates to include SCORM runtime tracking:

javascript

Copy

*// In your SCORM playback template*

const updateRuntime = async (data) => {

try {

const response = await fetch(`/scorm/runtime/${scormId}/update/`, {

method: 'POST',

headers: {

'Content-Type': 'application/json',

'X-CSRFToken': csrf\_token,

},

body: JSON.stringify(data)

});

const result = await response.json();

console.log('Runtime update:', result);

} catch (error) {

console.error('Runtime update failed:', error);

}

};