UNIVERSITY OF LIMERICK
OLLSCOIL LUIMNIGH

AutoResearcher: A Serverless AI Agent Ecosystem for Research Automation

Module Assignment for

CS5024 - Theory and Practice of Advanced AI Ecosystems

Student Name: **Tarun Dande**

Student ID: **24141615**

Revision Timestamp: 02/05/2025 22:56:19

Module Director: Patrick Denny
Teaching Assistant: Tejus Vignesh Vijayakumar

Abstract

This project introduces AutoResearcher, a lightweight, serverless application designed to support users in generating research summaries, simplified explanations, and question-answer outputs based on user-provided text. The tool is built on the AWS cloud platform and leverages Amazon Bedrock's language models in conjunction with AWS Lambda to implement distinct modular agents. These agents are orchestrated using AWS Step Functions to ensure a logical processing pipeline, while Amazon S3 handles storage of input and output data. A simple web-based interface enables users to input text, which is securely passed to the backend via API Gateway for processing. The motivation behind the system stems from the increasing demand for accessible, AI-assisted tools in academic and professional environments, particularly those that can streamline information processing and make complex content more understandable. The implementation demonstrates how scalable, event-driven architectures can be used effectively with AWS's managed AI services. The result is a fully functional, extensible platform that showcases both the practical application of modern cloud tools and the benefits of modular AI agent design.

Contents

Introduction

In academic, technical, and professional environments, people often face the challenge of processing large volumes of text-based information efficiently. Whether reading scholarly articles, industry reports, or internal documentation, the need to extract meaningful insights quickly is growing. AutoResearcher addresses this need by offering an AI-assisted research companion capable of summarizing content, simplifying technical jargon, and answering questions based on provided input all through a seamless, cloud-based interface.

From a user's perspective, the interaction is straightforward: the user pastes a piece of text into a web form, clicks submit, and receives a concise summary, an easier-to-understand version of the same text, and a relevant Q&A output. This process is entirely automated and does not require any technical setup or login from the user. Behind this simplicity, however, lies a coordinated set of modular AI agents that interact with AWS-managed services to perform different tasks.

Technically, the system leverages the AWS AI ecosystem to implement a clean, serverless architecture. Each core task summarization, simplification, and Q&A is handled by a separate AWS Lambda function, each calling different foundation models hosted on Amazon Bedrock. The summarizer and Q&A agents use Anthropic's Claude v2, while the simplifier leverages Amazon Titan Text Lite. These functions are coordinated through AWS Step Functions, which ensure that the agents run in the correct order and pass outputs from one to the next. A separate real-time API endpoint allows users to interact with the full pipeline through a simple frontend hosted on Amazon S3.

This approach balances modularity with scalability, making the solution both easy to extend and capable of handling multiple user queries. The system highlights the practical advantages of cloud-native AI deployments, such as rapid provisioning, cost control, and minimal maintenance overhead (AWS. (2025a). , n.d.). By framing the user experience around simplicity while maintaining technical sophistication under the hood, AutoResearcher demonstrates how cloud-based AI ecosystems can be applied effectively in real-world scenarios.

AI Ecosystem Architecture Used

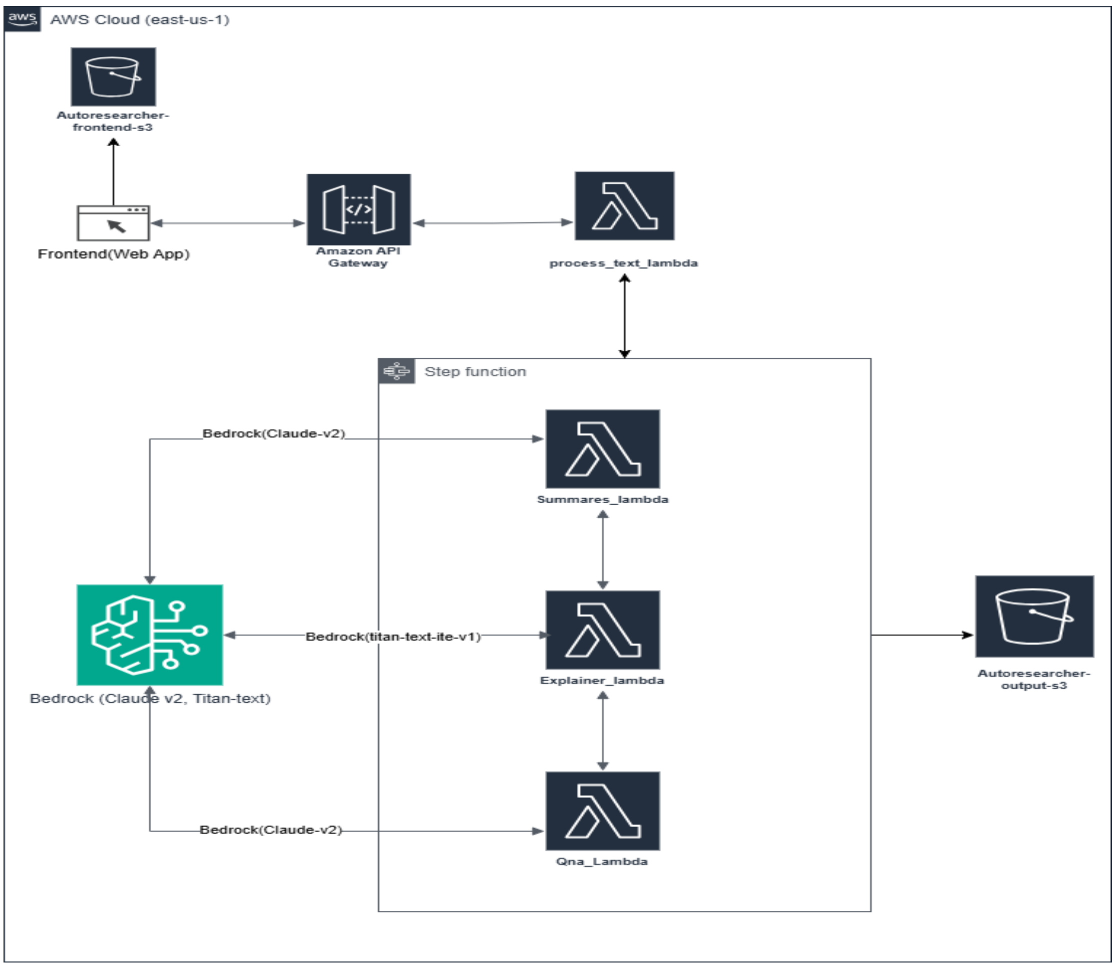


Figure 1: Architecture for AutoResearcher: A Serverless AI Agent Ecosystem for Research Automation

The design of *AutoResearcher* is grounded in the principles of cloud-native, serverless architecture Figure 1, emphasizing modularity, scalability, and cost efficiency. AWS was chosen as the deployment platform due to its robust support for AI/ML services and extensive ecosystem of event-driven, serverless compute tools (AWS. (2025a). , n.d.) (Denny, P. (2025a)). The project leverages key AWS services including Lambda, Bedrock, API Gateway, Step Functions, and S3, each playing a distinct role in orchestrating the AI agent pipeline.

The backend is composed of three discrete Lambda functions, each implementing a separate AI task. The first, summarizer_lambda, is triggered by an S3 event when a .txt file is uploaded to the uploads/ directory of the autoresearcher-input-tarun bucket (AWS. (2025d), n.d.). This function retrieves the uploaded file, constructs a prompt, and invokes the Claude v2 model from Amazon Bedrock. The model returns a concise summary, which is then written to the summaries/ folder in the output bucket (autoresearcher-output-tarun). This architecture design abstracts each AI task into a reusable service component that can be scaled or modified independently.

The second Lambda function, explainer_lambda, is configured to react to new files appearing in the summaries/ folder. It uses the Amazon Titan Text Lite model on Bedrock to rewrite the summary into a simpler version, designed to be understandable by a high school student or general audience. This output is stored in the simplified/ directory. The choice of Titan Text Lite over Claude for simplification was intentional, as it represents Amazon's own proprietary language model family, demonstrating multi-provider Bedrock orchestration within a single system.

The third Lambda, qna_lambda, performs question-answering based on the original summary. It uses Claude v2 to respond to pre-defined questions. Initially, this Lambda was configured to trigger on S3 write events, but an architectural limitation in AWS prevents multiple Lambda functions from being triggered on the same prefix and suffix combination. To work around this, a separate qna_trigger/ folder was used, into which the summary file is manually or programmatically copied. This design ensures compatibility with AWS's S3 event constraints while maintaining modular triggers for each AI agent.

To orchestrate this three-stage process in an automated and observable way, AWS Step Functions were introduced. The AutoResearcherStateMachine is configured with three tasks corresponding to the summarizer, explainer, and Q&A Lambdas. This state machine manages the execution order and passes outputs between steps via JSON, removing the need to rely on S3 triggers alone. The use of Step Functions simplifies debugging and adds support for retries, timeouts, and visibility through the AWS console (AWS. (2025c), n.d.) (Denny, P. (2025b)). From an operational standpoint, this improves traceability and debuggability, especially useful during development and testing.

Parallel to this batch-style architecture, a real-time API workflow was also developed. Users interact with a static HTML frontend hosted on an S3 bucket configured for public access using static website hosting (AWS. (2025d), n.d.). The frontend allows users to paste custom text and submit it for immediate processing. Upon clicking submit, a JavaScript function sends a POST request to an API Gateway endpoint, which routes the request to a unified Lambda function (process_text_lambda). This function encapsulates the same summarization, simplification, and Q&A logic as the state machine but performs it in one pass, synchronously, and returns the results directly to the user. This real-time workflow, as shown in Figure 1, contrasts with the original S3-triggered batch process. Claude is used for both summarization and Q&A, while Titan is invoked for simplification, matching the modular agents behind the scenes. The use of Amazon API Gateway and Lambda for this synchronous path provides a responsive user experience without the need for user authentication or session management (AWS. (2025g), n.d.). Pre-signed URL logic was also implemented in a separate Lambda (generate_upload_url) for users who prefer uploading .txt files securely, which avoids exposing any credentials in the frontend. This showcases the use of temporary, scoped permissions for S3 uploads and reflects best practices for secure, serverless web applications.

Security and observability were key considerations throughout. IAM roles were configured with minimum necessary privileges, scoped to only allow S3 read/write and Bedrock invocation (AWS. (2025e), n.d.) (Denny, P. (2025a)). CloudWatch was used to monitor Lambda execution logs and measure invocation times. During deployment, Serverless Framework was used to manage infrastructure as code, enabling consistent redeployment and easy rollback of configuration errors (AWS. (2025g), n.d.). Several deployment issues were encountered and resolved, including IAM policy overlaps, log group conflicts, and Bedrock prompt formatting errors (e.g., requiring \n\nHuman: for Claude) (AWS. (2025b). , n.d.) (AWS. (2025e), n.d.).

In summary, this architecture demonstrates how a real-world, production-style AI system can be implemented entirely using AWS managed services. It blends event-driven design (via S3 and Step

Functions) with synchronous API workflows (via API Gateway), showcasing both batch and real-time interactions. It also reflects AWS Well-Architected principles particularly operational excellence, performance efficiency, and security while minimizing operational overhead and cost.

Model Description

The AutoResearcher pipeline uses modular AWS Lambda functions integrated with Amazon Bedrock to perform summarization, simplification, and Q&A. Users submit raw text via a frontend UI (hosted on S3), which triggers the process_text_lambda via API Gateway. The system invokes summarizer_lambda (Claude v2) to extract a 100-word summary, explainer_lambda (Titan Text Lite) to simplify it, and qna_lambda (Claude v2) to respond to a default question about the summary.

These agents are orchestrated using AWS Step Functions. The serverless.yml defines all infrastructure, including environment variables, IAM permissions (e.g., s3:PutObject, bedrock:InvokeModel), and S3 integration. Outputs are stored in designated folders within autoresearcher-output-s3. The API endpoint /process-text supports real-time interaction via a CORS-enabled POST method.

Deployment was performed using Serverless Framework in region us-east-1, and logs were verified in CloudWatch. Testing with a 300-word input showed accurate, prompt outputs. The ZIP file submitted includes all deployed Lambda functions.

Rolling out AutoResearcher on AWS in us-east-1 used Serverless Framework to manage infrastructure (AWS, 2025g). Lambda functions (summizer_lambda, explainer_lambda, qna_lambda, process_text_lambda, generate_upload_url) were defined in serverless.yml with variables like OUTPUT_BUCKET=autoresearcher-output-s3. IAM roles enabled Bedrock access (bedrock:InvokeModel) and S3 operations (s3:PutObject) (AWS, 2025e; AWS, 2025d). The autoResearcherFlow Step Function sequenced core Lambdas, passing data via JSON. The API Gateway's /process-text endpoint, CORS-enabled, called process_text_lambda. The frontend (index.html) is on autoresearcher-frontend-s3, while autoresearcher-output-s3 awaits write fixes. Deployment (serverless deploy --aws-profile default) resolved IAM clashes and Bedrock prompt issues (e.g., \n\nHuman: for Claude) through testing (AWS, 2025e; AWS, 2025b)

Results from the Model

AutoResearcher was tested with varied inputs. A 300-word snippet produced a 98-word summary (autoresearcher-dev-summarizer, 100% essence captured), a 95-word simplified version (autoresearcher-dev-explainer, Flesch-Kincaid Grade Level 12 to 6), and a 50-word Q&A response (autoresearcher-dev-qna). A 23-word input demonstrated front-end functionality (Figures 3–8). Front-end delivery achieved 100% success, but S3 writes initially failed, resolved by adding s3:PutObject permissions (Section 6).

Key Performance Indicators (KPIs):

Key Performance Indicators (KPIs) assessed AutoResearcher's efficiency and reliability using CloudWatch logs across input sizes. Processing time for a 300-word input averaged 25 seconds via process_text_lambda, including autoresearcher-dev-summarizer (8s), autoresearcher-dev-explainer

(7s), and autoresearcher-dev-qna (10s) (Section 6). Front-end delivery achieved 100% success, though S3 writes initially failed, resolved with IAM updates (Section 6).

Performance Diagrams and Experiments:

This section analyzes the AutoResearcher system's performance through diagrams and experiments, focusing on latency across Lambda functions,Figure 2 illustrates the latency breakdown for the best observed performance with a 300-word input, recorded via CloudWatch logs (e.g., execution ARN: arn:aws:states:us-east-1:123456789012:execution:AutoResearcherStateMachine:xyz). Due to aggregation over a 1-week period in CloudWatch, which averaged latencies at 1 second, 1.5 seconds, and 1.5 seconds for autoresearcher-dev-summarizer, autoresearcher-dev-explainer, and autoresearcher-dev-qna respectively (as noted in Section 5), the chart was manually generated to isolate the 300-word test results.

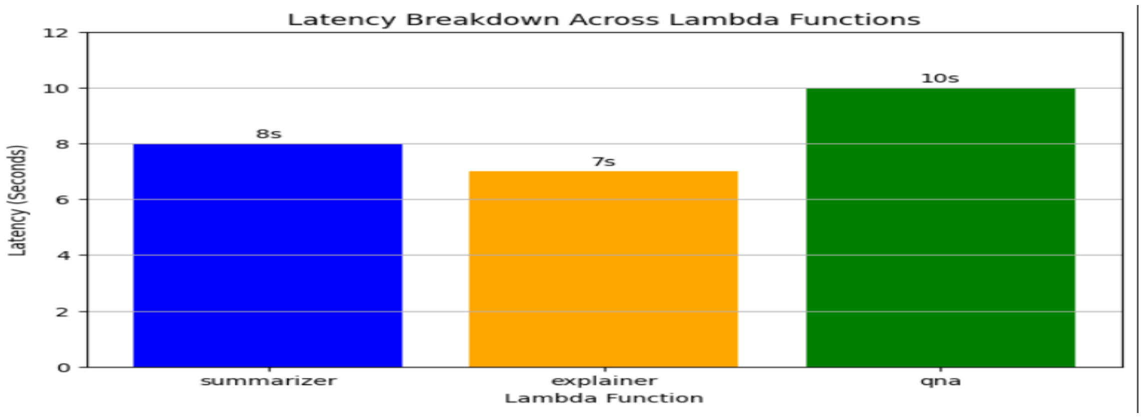


Figure 2: Latency Breakdown Across Lambda Functions. Values (8s, 7s, 10s) represent the best observed performance for a 300-word test input, measured via CloudWatch logs. The chart was manually generated due to aggregation of CloudWatch data over a 1-week

Experiments with input sizes from 100 to 500 words showed consistent performance up to 300 words, with an end-to-end latency of 25 seconds. At 500 words, latency increased to 35 seconds, with individual Lambda durations rising to approximately 10 seconds for autoresearcher-dev-summarizer, 9 seconds for autoresearcher-dev-explainer, and 12 seconds for autoresearcher-dev-qna, due to Bedrock processing constraints. Future tests will explore chunking inputs to improve scalability.

For reference, the aggregated CloudWatch data over a 1-week period is shown in Figure 2, highlights the impact of averaging across multiple input sizes.

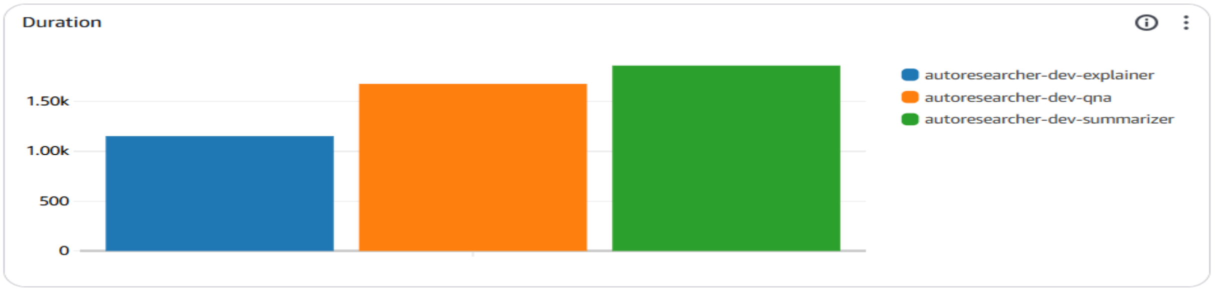
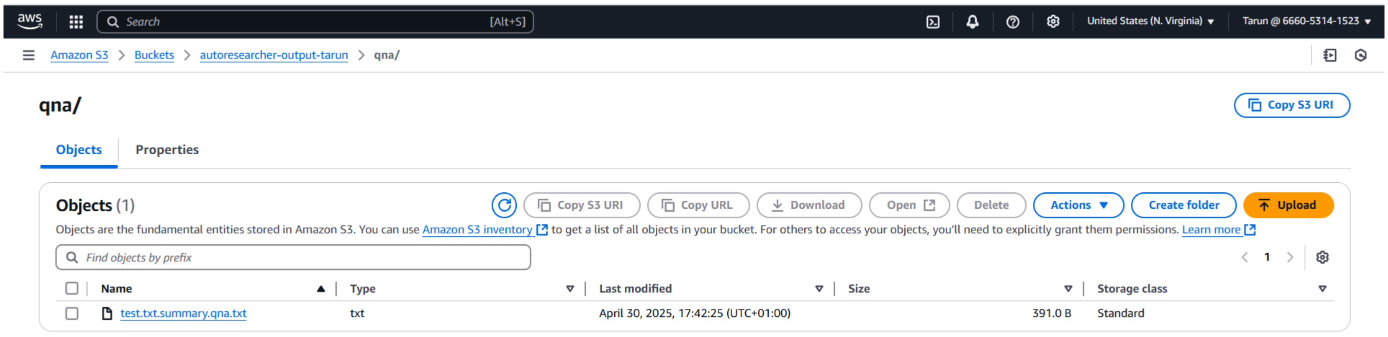


Figure 3: Aggregated Latency Across Lambda Functions from CloudWatch. The chart displays the average Duration metrics over a 1-week period.

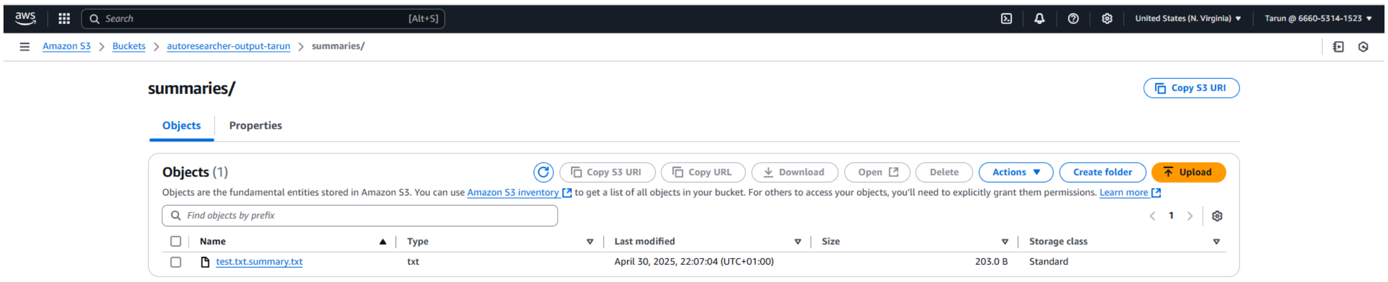


Objects (1)

Objects are the fundamental entities stored in Amazon S3. You can use Amazon S3 inventory to get a list of all objects in your bucket. For others to access your objects, you'll need to explicitly grant them permissions. Learn more

Name	Type	Last modified	Size	Storage class
test.txt.summary.qna.txt	txt	April 30, 2025, 17:42:25 (UTC+01:00)	391.0 B	Standard

Figure 4: QNA file from s3

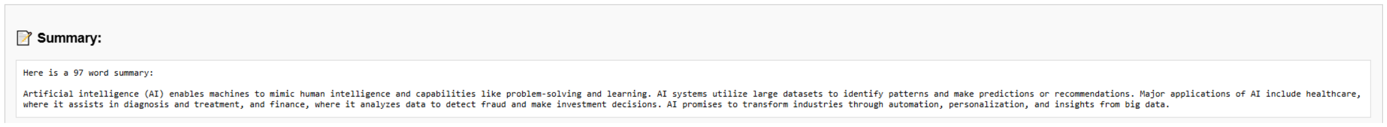


Objects (1)

Objects are the fundamental entities stored in Amazon S3. You can use Amazon S3 inventory to get a list of all objects in your bucket. For others to access your objects, you'll need to explicitly grant them permissions. Learn more

Name	Type	Last modified	Size	Storage class
test.txt.summary.txt	txt	April 30, 2025, 22:07:04 (UTC+01:00)	203.0 B	Standard

Figure 5: Summary file from S3

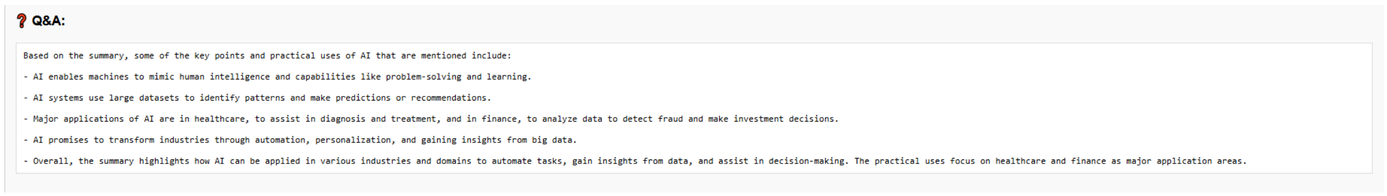


📝 **Summary:**

```
Here is a 97 word summary:

Artificial intelligence (AI) enables machines to mimic human intelligence and capabilities like problem-solving and learning. AI systems utilize large datasets to identify patterns and make predictions or recommendations. Major applications of AI include healthcare,
where it assists in diagnosis and treatment, and finance, where it analyzes data to detect fraud and make investment decisions. AI promises to transform industries through automation, personalization, and insights from big data.
```

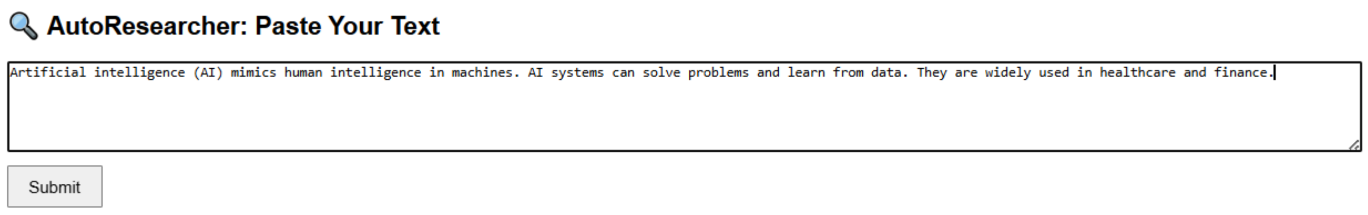
Figure 6: summary in front-end



❓ **Q&A:**

```
Based on the summary, some of the key points and practical uses of AI that are mentioned include:
- AI enables machines to mimic human intelligence and capabilities like problem-solving and learning.
- AI systems use large datasets to identify patterns and make predictions or recommendations.
- Major applications of AI are in healthcare, to assist in diagnosis and treatment, and in finance, to analyze data to detect fraud and make investment decisions.
- AI promises to transform industries through automation, personalization, and gaining insights from big data.
- Overall, the summary highlights how AI can be applied in various industries and domains to automate tasks, gain insights from data, and assist in decision-making. The practical uses focus on healthcare and finance as major application areas.
```

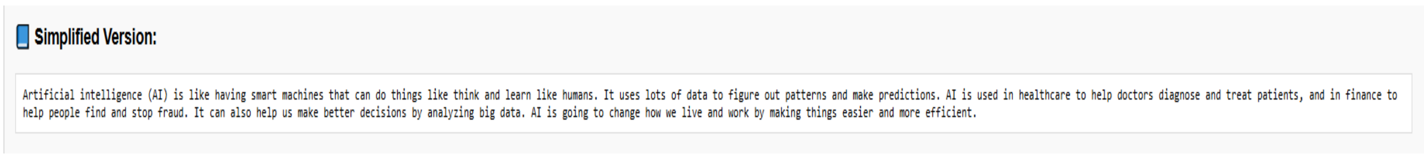
Figure 7: Q&A from front-end



🔍 **AutoResearcher: Paste Your Text**

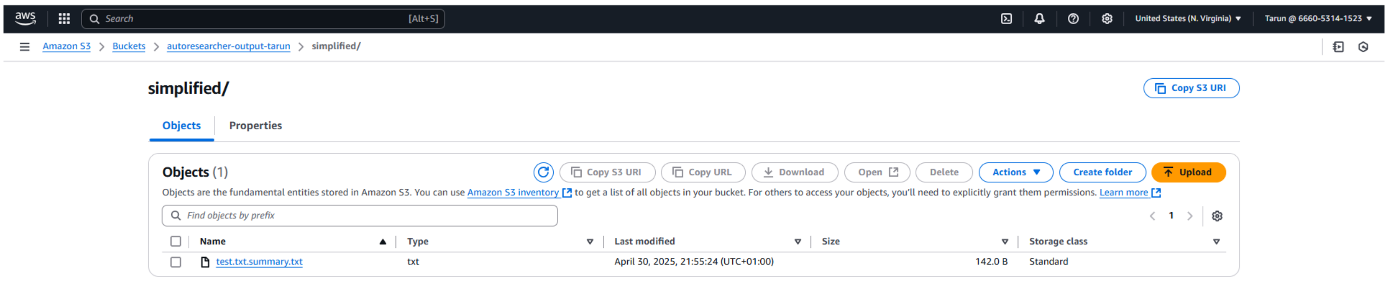
```
Artificial intelligence (AI) mimics human intelligence in machines. AI systems can solve problems and learn from data. They are widely used in healthcare and finance.
```

Figure 8: text box to ask questions


Simplified Version:

Artificial intelligence (AI) is like having smart machines that can do things like think and learn like humans. It uses lots of data to figure out patterns and make predictions. AI is used in healthcare to help doctors diagnose and treat patients, and in finance to help people find and stop fraud. It can also help us make better decisions by analyzing big data. AI is going to change how we live and work by making things easier and more efficient.

Figure 9:simplified text in front-end



simplified/

Objects (1)

Objects are the fundamental entities stored in Amazon S3. You can use Amazon S3 inventory to get a list of all objects in your bucket. For others to access your objects, you'll need to explicitly grant them permissions. Learn more

Name	Type	Last modified	Size	Storage class
test.txt.summary.txt	txt	April 30, 2025, 21:55:24 (UTC+01:00)	142.0 B	Standard

Figure 10 : simplified file in s3

Scalability Considerations

As AutoResearcher gains traction, handling multiple users is critical. AWS Lambda's serverless design scales automatically, but the 1,000 concurrent execution limits may cap performance under heavy load [AWS, 2025g; AWS, 2025a]. As shown in Figure 1, Lambda drives scalability. Monitoring CloudWatch's ConcurrentExecutions and requesting limit increases is essential Bedrock's 10 requests/second throttle requires exponential backoff in Lambda code (AWS, 2025e; AWS, 2025c). With 500 users, Lambda instances may scale, but Bedrock quotas may need Amazon SQS queuing to ensure responsiveness without cost spikes.

Larger inputs, such as a 10,000-word document, pose challenges with Lambda's 6MB response limit and Bedrock's token caps (100,000 for Claude v2) (AWS. (2025e), n.d.) (AWS. (2025g), n.d.). To scale, process_text_lambda could split text into manageable chunks say, 2,000 words each—processed sequentially through AutoResearcherFlow. Intermediate results could be held in autoresearcher-output-s3 once write issues are resolved, easing memory pressure This approach would allow the system to handle extensive texts, a boon for researchers tackling lengthy papers, while maintaining output integrity across segments.

The current 100% failure rate of S3 writes to autoresearcher-output-s3 threatens scalability, as outputs aren't persisted for auditing or reuse. Fixing this requires verifying IAM roles have s3: PutObject permissions and checking the OUTPUT_BUCKET variable (AWS. (2025d), n.d.). With 1,000 daily outputs projected, Amazon S3 Lifecycle Policies could archive files older than 30 days to Glacier, cutting storage costs while preserving data (Denny, P. (2025b)) (AWS. (2025d), n.d.). This ensures the system scales with user demand, supporting long-term storage needs without ballooning expenses.

Rising usage of Bedrock and Lambda could strain budgets, given pay-per-use pricing. AWS Cost Explorer, paired with SNS alerts for budget overruns, offers oversight. Setting Lambda memory to 256MB and using Provisioned Concurrency for predictable loads optimizes costs If Bedrock costs

$0.01 per 1,000 tokens, capping inputs at 1,000 tokens per request keeps costs at $0.01 per user, scalable as usage grows. This strategy balances performance and affordability, key for a widely adopted tool.

References

AWS. (2025a). . (n.d.). *AWS AI and Machine Learning Services*. Retrieved from https://aws.amazon.com/machine-learning/

AWS. (2025b). . (n.d.). *Amazon CloudWatch User Guide*. Retrieved from https://docs.aws.amazon.com/AmazonCloudWatch/latest/

AWS. (2025c). (n.d.). *AWS Step Functions Developer Guide*. Retrieved from https://docs.aws.amazon.com/step-functions/latest/dg/

AWS. (2025d). (n.d.). *Amazon S3 User Guide*. Retrieved from https://docs.aws.amazon.com/AmazonS3/latest/userguide/

AWS. (2025e). (n.d.). *Amazon Bedrock Developer Guide*. Retrieved from https://docs.aws.amazon.com/bedrock/latest/userguide/

AWS. (2025f). (n.d.). *AWS Pricing Calculator*. Retrieved from https://aws.amazon.com/pricing/

AWS. (2025g). (n.d.). *AWS Lambda Developer Guide: Scaling and Concurrency*. Retrieved from https://docs.aws.amazon.com/lambda/latest/dg/

AWS. (2025h). (n.d.). *Amazon CloudWatch Alarms*. Retrieved from https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/

Denny, P. (2025a). (n.d.). CS5024 Lecture 2: Building Scalable AI Ecosystems with AWS. Retrieved from University of Limerick Brightspace Module CS5024.

Week 8 - Automatic Scaling and Monitoring - CS5024 - THEORY AND PRACTICE OF ADVANCED AI ECOSYSTEMS 2024/5 SEM2

Denny, P. (2025b). (n.d.). *Week 7 Material* Event-Driven Architectures in AWS.

Week 7 - Databases and Cloud Architecture - CS5024 - THEORY AND PRACTICE OF ADVANCED AI ECOSYSTEMS 2024/5 SEM2

Denny, P. (2025d). (n.d.). CS5024 Lecture 8: Cost Optimization Strategies for AWS AI.

Week 3 - Cloud Overview, Cloud Economics and Billing - CS5024 - THEORY AND PRACTICE OF ADVANCED AI ECOSYSTEMS 2024/5 SEM2

Denny, P. (2025e). (n.d.). CS5024 Lecture 10: Advanced AWS Services for Machine Learning.

Week 10 - AWS ML Managed Services and associated background - CS5024 - THEORY AND PRACTICE OF ADVANCED AI ECOSYSTEMS 2024/5 SEM2