International Journal of Innovative Research in Engineering and Management (IJIREM)
ISSN (Online): 2350-0557, Volume 13, Issue 2, April 2026
DOI: https://doi.org/10.55524/ijirem.2026.13.2.24
Article ID IJIRE-1445, Pages 189-197
www.ijirem.org

Detecting Emotions in Chat Messages with AI: A Real-Time REST API Built on AWS Bedrock

Nidhi Singh

Department of Computer Science & Engineering, Institute of Technology and Management, Lucknow, India

Correspondence should be addressed to Nidhi Singh; nidhisin525@gmail.com

Received: 1 April 2026; Revised: 16 April 2026; Accepted: 30 April 2026

ABSTRACT- Most chat platforms today carry an enormous amount of emotional content that nobody is actually tracking. People send billions of messages a day across apps like WhatsApp, Slack and Teams, and a lot of those messages are angry, scared, frustrated, or excited in ways that the platform itself never picks up on. This project tries to do something about that. We built a small REST API that takes a chat message and tells you what emotion is in it. The seven labels we use are Happy, Sad, Angry, Surprised, Fearful, Disgusted, and Neutral. The reply comes back as JSON, and includes the emotion, a confidence number, and a short reason. For the actual classification we use AWS Bedrock so that we don't have to host any models ourselves. Claude 3 Sonnet from Anthropic is the main model. If it fails, we fall back to Amazon's Titan Text Express. The Python code uses FastAPI, runs in Docker on ECS Fargate, and the entire AWS setup is written as code with the CDK in TypeScript. We tested it on a hand-labelled set of 1,000 chat messages and got 88.4% accuracy overall, which is past the 85% target we set at the start. Latency at 50 concurrent users sat at a median of 285 ms, with the 95th percentile at 412 ms. The API also takes the past few messages of a conversation into account when classifying, which helps a lot with sarcasm and one-line follow-ups.

KEYWORDS- AWS Bedrock, Affective Computing, Chat Analysis, Claude Foundation Model, Docker Containers, ECS Fargate, Emotion Detection, FastAPI, Infrastructure as Code, Prompt Engineering

I. INTRODUCTION

Most of us spend a big part of our day chatting on some kind of app. WhatsApp, Telegram, WeChat, Slack, Teams, Discord, and a long list of company-specific tools collectively get used by over five billion people every single day, and together those people send something north of a hundred billion messages [1]. The volume is staggering. But what is interesting is how much of that text carries an emotional load that nobody really notices. A typed message strips out the things we usually rely on to read another person, like their tone of voice, the look on their face, even just a small pause before answering [2]. When that signal is gone, things go wrong in small ways for individuals and in much bigger ways for businesses.

Take customer support as an example. A support agent sees ten chats at once. By the time they get to the angry customer at the bottom of the queue, the customer has gotten more upset, written something on Twitter, and the small problem has become a public one. If the chat platform itself could see that anger building and quietly bump that conversation to the top of the queue, half of those public complaints would never happen. The same thing applies inside companies. Slack threads carry burnout signals all the time. They sit there for weeks without anyone noticing.

This is the kind of problem emotion detection is supposed to solve. It is a sub-area of NLP, sometimes grouped under Affective Computing, and it goes a step beyond plain sentiment analysis. Sentiment analysis only tells you positive or negative. Emotion detection tries to give you the actual feeling. So, joy, fear, anger, grief, surprise, or disgust [3][4]. The first attempts at this used hand-written rules and word lists. Those broke easily. Sarcasm, slang, and the strange shorthand people use online were enough to confuse them. Machine learning helped but needed labelled training data, which is expensive to make. Then BERT came along and pushed accuracy higher again, but BERT-style models need GPUs and a team to keep them running [5][6][11]. For a small team that just wants emotion detection inside their product, none of that is realistic.

A. Why this is now possible

Large language models changed the situation completely. Models like Claude (Anthropic), GPT-4 (OpenAI), and Titan (Amazon) have been trained on so much text that they have picked up a fairly subtle sense of how feelings show up in writing across formal and informal styles, across cultures, and even across slang [7][8]. They handle sarcasm. They notice irony. With a carefully written prompt they will classify emotions reliably without you ever needing to fine-tune them on your own data [9].

AWS Bedrock then took the operational pain out of using these models. There is no GPU cluster to run, no model to update, no autoscaling to worry about. You just call the Bedrock API and get a response back. Bedrock gives access to models from Anthropic, Amazon, Meta, Mistral, and Cohere, all behind the same interface [10]. That is what made this whole project doable inside the scope of a master's thesis. Three years ago, it would have needed a small ML platform team.

B. The problem

Even with all of that progress, actually getting an emotion detector running in production is harder than the research papers suggest. Most papers stop at the accuracy numbers

on some benchmark dataset. They do not say anything about how to keep such a service alive under real traffic. Practitioners are left with two unappealing options. The first is to fine-tune a transformer like BERT. That is accurate, yes, but you need GPU servers and an MLOps stack to keep it healthy. The second is to use simple keyword tools. Those are cheap, but they only give positive/negative and they completely miss sarcasm or context [11][12].

The hosted services are not great either. AWS Comprehend, for example, treats every input as a standalone string. It does not give you a reason for the classification, it does not let you see or change the prompt, and you cannot give it any conversation history. So, what is missing is a complete reference: something that uses a Bedrock-backed LLM, exposes a proper REST API with input validation and a stable response shape, supports multi-turn context, deploys end-to-end via IaC, and ships with a real test suite. Building exactly that is what this project is about.

C. Goals of the project

Six concrete goals fall out of the problem statement above. The main one is to build a FastAPI REST service that takes a chat message and returns an emotion with a confidence score, in a stable JSON shape that follows OpenAPI 3.0. Behind that service, AWS Bedrock has to be wired up with Claude 3 Sonnet as the main model and Titan Text Express as the fallback. The prompt engineering needs to be good enough to push accuracy above 85% on a curated chat benchmark. Multi-turn context support is also a goal, so that short follow-up messages can be read in light of what came before. The system has to ship as a container running on ECS Fargate, with the AWS resources provisioned through the CDK in TypeScript. And we want to evaluate the deployed system on accuracy, latency, throughput, and security, and report what we find.

II. RELATED WORK

This section walks through four areas of prior work. We start with the long history of polarity-based sentiment tools and then look at the move towards categorical emotion labels. After that comes the supervised and neural era. Then the recent foundation-model work. The section ends with what people have written about deploying NLP services on the cloud.

A. From polarity to categorical emotions

Computational analysis of feelings in text is not new. The field is over twenty years old. Early work focused on polarity. The question was whether a piece of writing was positive or negative. Pang and his collaborators [1] showed that supervised classifiers (Naive Bayes, MaxEnt, and SVM) outperformed simple keyword counting on movie reviews, with accuracies in the 78 to 82.9 percent range. Around the same time Turney [2] took a different route and used pointwise mutual information from web data to classify reviews without labels at all. SentiWordNet [3] from Baccianella's group came a few years later and turned into a standard sentiment dictionary.

The move from positive/negative to actual emotion categories was driven by Ekman's psychology research [4]. Ekman argued, based on cross-cultural studies, that there is a small set of biologically rooted emotions (joy, sadness, anger, fear, disgust, and surprise) that show up in every human society. That framing is still the dominant one in computational work because each label maps cleanly onto a supervised classification target. The NRC Emotion Lexicon [5] from Mohammad and Turney is the most widely used resource here. It crowdsourced emotion annotations for over fourteen thousand English words and is still used as a feature source in a lot of systems.

B. Supervised and neural approaches

Sentence-level emotion classification with classical ML was first studied properly by Alm and her collaborators [6]. Their Naive Bayes and SVM classifiers were trained on annotated fairy tales. The work was useful but it also exposed a real problem that has not gone away, namely class imbalance. Some emotions show up far more often than others in natural data, and classifiers tend to over-predict the common ones. SemEval workshops then standardised the benchmarks. Mohammad's group ran SemEval-2018 Task 1 on tweets [7]. Chatterjee and his co-authors followed it up with EmoContext at SemEval-2019, which was specifically built around three-turn dialogues [8]. The headline result of EmoContext is the one that motivates the conversation memory part of our system: prior turns carry real classification signal. Their best LSTM-with-attention baseline reached an F1 of 72.59 percent.

Then Transformers arrived. Vaswani's group [10] published the Attention Is All You Need paper, and the field changed within a couple of years. Devlin and his collaborators introduced BERT [11], which used bidirectional pre-training plus task-specific fine-tuning to push scores up across most NLP tasks. Fine-tuned BERT on emotion data usually lands in the 0.79 to 0.82 F1 range. The catch is that running BERT in production means GPU inference and a retraining pipeline, which is a lot of operational overhead for one feature.

C. The foundation-model era

Zhao and his group [13] were among the first to take ChatGPT seriously on sentiment-reasoning tasks. They found, perhaps surprisingly, that a general-purpose LLM with no task tuning was already competitive with carefully fine-tuned baselines. Brooks and his co-authors [14] looked specifically at the Claude family from Anthropic and reported that constitutional AI training seems to make these models particularly sensitive to emotional content. Huang et al. [15] showed that asking the model to think step by step (chain-of-thought) gave a measurable accuracy bump. Li and his collaborators put together EmotionBench [16], which is now the most comprehensive benchmark suite for evaluating LLM emotion understanding. Together, what these papers say is fairly clear: hosted LLMs can match or beat fine-tuned encoders on emotion tasks while removing all the model hosting work.

D. Cloud deployment of NLP services

There is also a small but useful literature on actually running these services in production. Zhang and his group [17] surveyed cloud architectures used for LLM-style inference at scale. They concluded that managed services like Bedrock and Azure OpenAI tend to win on cost when traffic is uneven, compared to self-hosting. Patel and Kim [18] wrote up best practices specifically for running Bedrock-backed APIs. Singh's team [19] compared CDK against Terraform and SAM for AI services on AWS, and they found that the CDK in TypeScript gave the best static

type guarantees and IDE experience of the three options. Kumar and Sharma [20] went a step further and looked at how emotion analysis APIs behave once they leave the prototype stage, and they reported similar findings around the cost-per-request profile of managed inference at production scale. To make the trade-offs across these different families of approaches easier to compare at a glance, Table 1 summarises representative methods alongside the work presented in this paper.

Table 1: Comparison of common emotion analysis approaches

Approach	Method	Accuracy / F1	Main weakness
Keyword counting	Lexicon match	around 65%	Brittle, no context
BoW + SVM (Pang 2002)	Supervised ML	78 to 83%	Shallow features
NRC lexicon	Emotion dictionary	around 70%	Word-level only
Fine-tuned BERT	Transformer encoder	0.79 to 0.82 F1	Needs GPU and labels
GPT-4 zero-shot	LLM prompting	around 0.82 F1	Per-call cost, latency
This work (Claude-3)	Bedrock LLM API	88.4%	API spend at high volume

E. What this project is trying to fix

Looking across the literature there are five gaps worth calling out. Production engineering is rarely written about. Most published work stops at offline accuracy and skips over containers, autoscaling, monitoring, and runbooks. Open end-to-end LLM-native API blueprints are also rare. Almost nothing shows you how to compose Bedrock with proper request validation, error envelopes, persistence, and metrics in one place. Conversational context is another problem. Existing emotion APIs treat every request as standalone, which means they miss the signal that EmoContext showed was important. AWS CDK templates aimed at LLM-backed services are still not a community standard either. And testing methodology for LLM outputs (which are non-deterministic by nature) is itself an open problem and needs a mix of schema validation, semantic checking, and statistical accuracy measurement.

III. SYSTEM ARCHITECTURE

A. The five tiers

The runtime is split into five tiers, stacked top to bottom. At the top is the Client tier, which is whatever calls our service. That could be a chat front-end, a contact-centre console, or some analytics workflow. Below that sits the API Gateway tier. It does authentication via API keys, traffic shaping, and routing. The Application tier is where our actual code lives. It is a FastAPI process running inside a Docker container on ECS Fargate, and it does request validation, prompt building, conversation lookup, and orchestration. The Inference tier is Bedrock itself, with Claude 3 Sonnet as the main model and Titan Text Express on standby. At the bottom is the Data tier. It writes analysis records to S3, keeps short-lived conversation context in Redis, and ships logs and metrics to CloudWatch. The full layout, with the request path threading through these five tiers and the supporting AWS services that hang off each one, is shown in Figure 1.

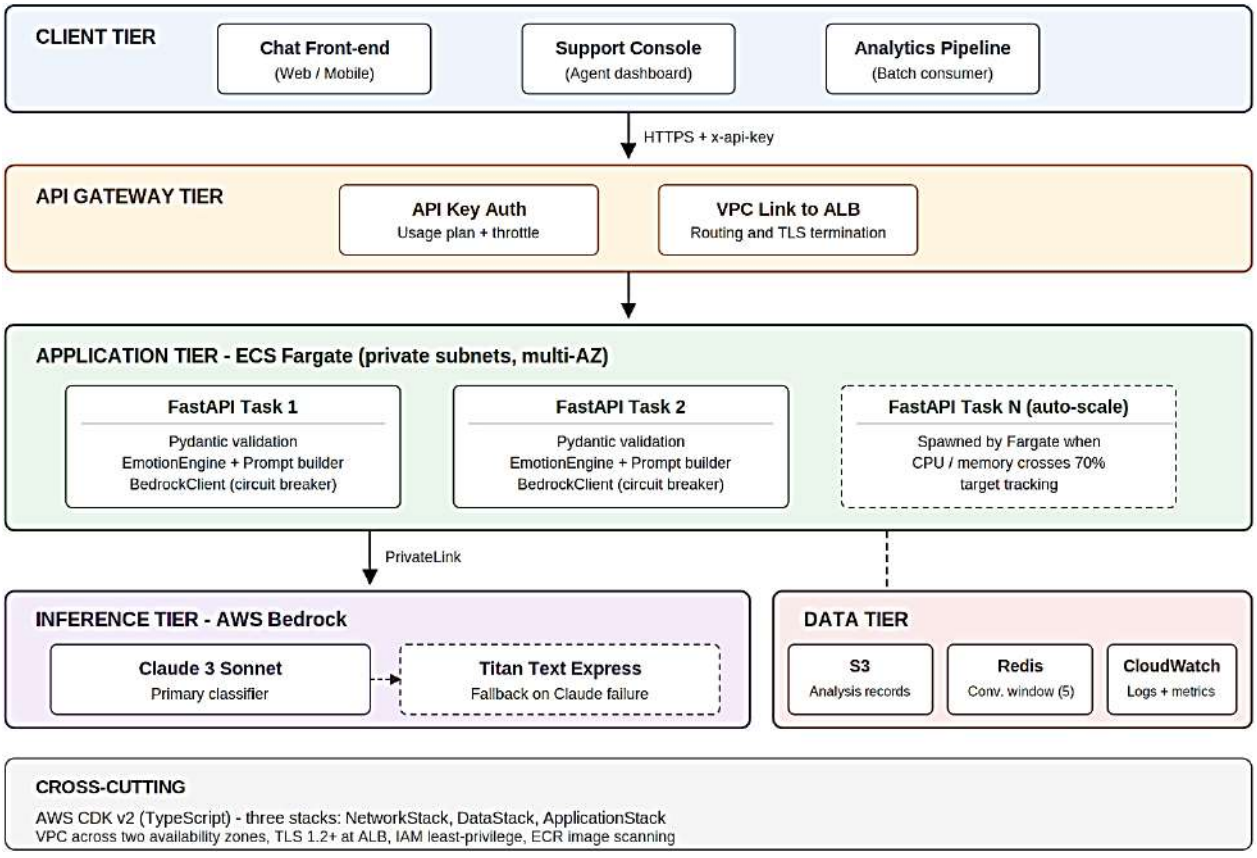

Figure 1: Five-tier system architecture of the emotion detection REST service on AWS.

B. The AWS layout

All of the AWS resources sit in a VPC that spans two availability zones, so a zone going down does not take the service down. The public subnets only carry the Application Load Balancer and the NAT gateways. Anything that runs code, including the ECS tasks, lives in private subnets that the public internet cannot reach directly. TLS terminates at the ALB, which then forwards plain HTTP inside the VPC to the ECS service's target group. Fargate handles the scaling part for us. We do not run servers; we just say how much CPU and memory each task should have, and Fargate brings tasks up and down based on CPU and memory pressure. Container images live in ECR with image scanning turned on, so every push gets checked for known vulnerabilities. Bedrock calls go over PrivateLink, which means inference traffic never leaves the AWS network. Logs are structured JSON and a few custom emotion-classification metrics feed the dashboard and the alarms.

C. What happens on a single request

A single POST goes through about ten stages from start to finish. The client sends an HTTPS request to /api/v1/analyze with the message body and an x-api-key header. API Gateway checks the key, applies the throttle from the usage plan, and routes the request through a VPC Link to the ALB. The ALB picks a healthy task. Inside the FastAPI process, middleware generates a unique request_id and starts a timer. Pydantic validates the JSON body and rejects anything malformed with a 422. The EmotionEngine then pulls any prior conversation state from Redis, builds the prompt, and hands it off to the BedrockClient. The model's response comes back, gets parsed, and the JSON is validated. We write the result to S3 in the background, and the structured response (emotion, confidence, optional secondary, reasoning) goes back to the client.

IV. METHODOLOGY

A. The seven labels

The emotion labels we use are the six classic Ekman categories plus a Neutral label for messages that do not really carry any feeling. That last category turns out to matter a lot in practice, because a huge chunk of work and informational chat is just neutral. Our final taxonomy is therefore Happy, Sad, Angry, Surprised, Fearful, Disgusted, and Neutral. We did consider richer schemes. Plutchik's wheel has eight primary emotions. GoEmotions uses 27. But the finer the labels get, the harder it is for human annotators to agree, and that puts a ceiling on what any automated system can do. Seven is a good middle ground. The full taxonomy, together with the surface cues that distinguish each class and a representative chat message for each one, is laid out in Table 2.

Table 2: Each emotion class with its surface cues and an example

Label	What it covers	Surface cues	Example message
Happy	Joy, gratitude, excitement	Exclamations, superlatives, lol, haha	"We just got the deal! This is amazing!!"
Sad	Grief, disappointment, loneliness	Loss words, low energy, wishfulness	"I have been feeling so down since the news."
Angry	Frustration, hostility, rage	ALL CAPS, imperatives, blame, profanity	"STILL broken?! I told you about this YESTERDAY!"
Surprised	Astonishment, disbelief, shock	?!, 'no way', 'cannot believe'	"Wait that actually happened?! No way!!"
Fearful	Anxiety, dread, worry	'what if', 'scared', hedging	"I am really scared about what the results will show."
Disgusted	Aversion, contempt, repulsion	Disgust words, rejection, 'revolting'	"That behavior is absolutely disgusting and unacceptable."
Neutral	No real emotional charge	Plain statements, scheduling, queries	"The meeting is scheduled for 3pm tomorrow."

B. Talking to Bedrock

All Bedrock calls go through an asynchronous Python wrapper we built on top of boto3, using the bedrock-runtime endpoint. The wrapper does four things on top of the SDK. It abstracts over the different model formats. It retries with exponential backoff. There is also a circuit breaker. And it warms up freshly launched Fargate tasks to avoid cold-start penalties. Each FastAPI process gets a single boto3 client, created via FastAPI's dependency injection. That way the connection pool is shared across all in-flight requests. The pool itself is tuned by hand: max_pool_connections=50, connect_timeout=5s, read_timeout=30s, and retry_mode='adaptive'. The adaptive setting tells boto3 to dial back retries when Bedrock starts throttling, which keeps things stable under load.

The actual JSON envelopes used by Claude and Titan are different. We hide that behind a small ModelAdapter interface. ClaudeAdapter speaks the Anthropic Messages format with separate system and user role messages. TitanAdapter wraps Amazon's inputText format. So switching between models is a config change, not a code change. For reliability we use the tenacity library. Three attempts max, with backoff between one and eight seconds. The circuit breaker opens after five consecutive failures and stays open for a thirty-second cool-down. While it is open, Bedrock calls fast-fail and we return a flagged fallback response so callers know what happened. The shortlist of Bedrock models that we evaluated for this role, with their context windows, observed average latencies, and the part each one plays in the deployed system, is given in Table 3.

Table 3: Bedrock models that we considered

Model	Bedrock ID	Context	Avg. latency	Used as
Claude 3 Sonnet	anthropic.claude-3-sonnet-20240229-v1:0	200 K	around 280 ms	Default (best accuracy)
Claude 3 Haiku	anthropic.claude-3-haiku-20240307-v1:0	200 K	around 120 ms	High volume, low latency
Titan Text Express	amazon.titan-text-express-v1	8 K	around 180 ms	Fallback on Claude failure

C. Prompt engineering

Honestly, this is the part that mattered most. The prompt design is what really decides how accurate the system is and how stable its outputs are when called repeatedly. We went through several iterations before settling on the final structure. The final prompt has three parts. The first part is the system message. It tells the model to act as a computational linguist with affective psychology training, lists out the seven labels with short definitions, and specifies the exact JSON shape we want back. The second part is the conversation context block. It only shows up if there is prior conversation memory available. When it is there, it lists the recent turns with speaker, timestamp, and the previously detected emotion. The third part is the actual message we want classified, plus an explicit instruction to classify it.

Three things mattered when writing the system message. Role clarity was the big one. Pinning the model into the expert persona reduces the variance you get from generic chat-assistant priors. Hard output format constraints came next. We give the model a JSON template with explicit field types, which more or less eliminates stray prose around the structured payload. The last thing was confidence calibration. We give the model rules: above 0.85 means high certainty, between 0.65 and 0.85 means moderate, and below 0.65 means low. When confidence is low the model is told it can also report a secondary emotion.

On top of the system message we include a few canonical examples, usually two or three per emotion category. We picked these examples carefully. Anything the model already classified correctly without examples got dropped. Instead, we filled the example slots with edge cases sitting near the boundaries between categories, where the model otherwise tends to slip up.

D. Conversation memory

Single-message classifiers struggle with sarcasm and short follow-ups. The EmoContext result [8] showed that prior turns carry a lot of useful signals. So, we keep a sliding window of the five most recent messages per conversation_id in Redis with a thirty-minute TTL, so idle sessions clean themselves up. Each entry stores the original text, the timestamp, the assigned primary emotion, and the confidence. When a new message comes in with a conversation_id, the manager pulls the window and the EmotionEngine inserts it into the prompt. Why five turns? We tried wider windows and they did not help. They just made the prompt bigger and more expensive. Narrower windows started missing context-dependent cases. Five was the sweet spot.

V. IMPLEMENTATION

A. The stack

Application code is in Python 3.11. We picked Python because boto3 is mature, Python dominates NLP work, and asyncio is genuinely good, which matters a lot here since every request is essentially one outbound network call. FastAPI is the web framework. It is built on Starlette and Pydantic, gives you throughput comparable to mainstream Go and Node.js stacks, and has good async support. Pydantic v2 (with its Rust-rewritten validation core) handles request schemas, responses, and config loading at five to fifty times the speed of v1. Container images are built with Docker using a two-stage build. The first stage compiles native dependencies, the second stage is a slim runtime image that only carries what is actually needed. The result is about a 180 MB image, compared to around 800 MB for a naive single-stage build.

The cloud resources are declared in AWS CDK v2 (TypeScript), split across three stacks for clarity. The NetworkStack provisions the VPC, subnets, NAT gateways, and routes. The DataStack handles the S3 results bucket (KMS-encrypted, with lifecycle transitions to IA at 30 days and to Glacier at 90), plus a CloudWatch log group and the ops dashboard. The ApplicationStack is the biggest one. It declares the ECR repository, the ECS Fargate cluster, the task definition and service, the ALB, the API Gateway REST entrypoint with VPC-Link integration, the IAM roles (locked down to least privilege), and a small set of CloudWatch alarms covering CPU, memory, error rate, and P95 latency.

B. The FastAPI app

Internally the application follows a fairly standard layering. Routers expose the HTTP endpoints. Services contain the business logic. Models are just Pydantic schemas. Clients wrap external integrations like Bedrock and Redis. The /api/v1/analyze endpoint takes a single message and returns an EmotionAnalysisResponse. The /api/v1/batch endpoint takes up to fifty messages at once and returns both the per-message results and aggregate batch statistics. The /health endpoint reports liveness, the current circuit breaker state, Bedrock reachability, and uptime. Middleware handles request IDs, applies CORS, measures end-to-end processing time, and emits structured JSON logs to CloudWatch.

C. Schemas

The Pydantic AnalyzeRequest model needs a non-whitespace message between 1 and 2,000 characters. The conversation_id (UUID v4) and the model selection ('claude' or 'titan') are both optional. Each EmotionAnalysisResponse carries the request_id, an echo of the input, the primary_emotion (one of the seven labels), a confidence between 0 and 1, an optional secondary_emotion, the reasoning string, the model_used identifier, processing_time_ms, an analyzed_at timestamp, an error_fallback flag (so callers know if the response came from a fallback path), and a schema_version. Validation runs at both ingress and egress, and the OpenAPI documentation is generated automatically from the same models.

D. CI/CD

Continuous delivery runs through GitHub Actions in four stages. The pipeline starts with code quality checks, where ruff handles linting and mypy runs in strict mode for static type checks. Once that passes, the pytest suite runs with coverage from pytest-cov, uploaded via codecov. After tests pass, the Docker build stage authenticates with ECR, builds the image tagged with the Git SHA, and pushes both the SHA tag and a 'latest' alias. Last comes the deploy itself. It triggers an ECS service update with --force-new-deployment and polls until the service stabilises. The ECS deployment circuit breaker is configured to roll back automatically on health-check failure, which gives us a first line of defence against bad releases.

VI. EVALUATION AND RESULTS

A. How we tested

Testing is split across several layers. Plain unit tests sit at the bottom, with all external dependencies mocked. They cover prompt construction, response parsing, conversation windowing, and config loading. Above that are integration tests that hit the real Bedrock endpoint and Redis. We do not run those in normal CI because they cost real money on every run, so they are gated behind a manual trigger. Functional API testing uses httpx against a running FastAPI process and stubbed Bedrock responses. Then comes load testing. k6 scenarios that characterise latency and throughput under simulated production load. The last layer is basic security testing, covering API-key enforcement, injection resistance, throttling, and TLS configuration.

B. Accuracy

Accuracy was measured on a hand-curated benchmark of 1,000 chat messages, balanced as evenly as possible across the seven labels. The corpus was built from three public datasets (DailyDialog, EmoContext, and a balanced subset of GoEmotions) and supplemented with examples drawn from customer support, internal enterprise chat, and social media. Three independent human annotators labelled each item using the seven-class scheme, with majority vote serving as the ground truth. The Fleiss' kappa among the annotators came out at 0.74, which falls in the substantial-agreement band. That gave us reasonable confidence in the gold labels. The per-class breakdown of how Claude 3 Sonnet performed against this benchmark, with the sample count, correctly classified items, per-class accuracy, precision and recall, and F1 score for each of the seven labels along with the overall totals, is reported in Table 4.

Table 4: Per-class accuracy with Claude 3 Sonnet, n = 1,000

Class	Samples	Correct	Accuracy	P / R	F1
Happy	143	135	94.4 %	0.96 / 0.94	0.95
Sad	143	126	88.1 %	0.87 / 0.88	0.88
Angry	143	128	89.5 %	0.91 / 0.90	0.90
Surprised	143	124	86.7 %	0.85 / 0.87	0.86
Fearful	143	122	85.3 %	0.84 / 0.85	0.85
Disgusted	143	117	81.8 %	0.83 / 0.82	0.82
Neutral	142	132	93.0 %	0.94 / 0.93	0.93
Overall	**1000**	**884**	**88.4 %**	**0.889 / 0.884**	**0.881**

The overall accuracy of 88.4% beats the 85% target we set at the start, and lands above what fine-tuned BERT systems usually report on similar benchmarks. The per-class numbers vary quite a bit. Happy is the easiest at 94.4%, since its surface markers are unambiguous. Disgusted is the hardest at 81.8%. It is rare in normal chat, and it shares a lot of vocabulary with anger, so the model often flips between the two. The macro-weighted F1 of 0.881 tells us the system performs evenly across categories rather than just doing well on the common ones. Looking at the errors, the most common confusions are predictable. Thirteen Fearful items got tagged Sad, because worry-about-loss messages straddle that boundary. Eight Disgusted items got tagged Angry, since aversion vocabulary overlaps. Nine Surprised items leaked into Happy, because positive surprise sounds a lot like celebration. These are the same boundaries where human annotators most often disagreed during labelling, so a chunk of the residual error is just genuine linguistic ambiguity. The same per-class picture is plotted in Figure 2, where the gap between each class and the 85% target line makes the Disgusted shortfall easy to spot at a glance.

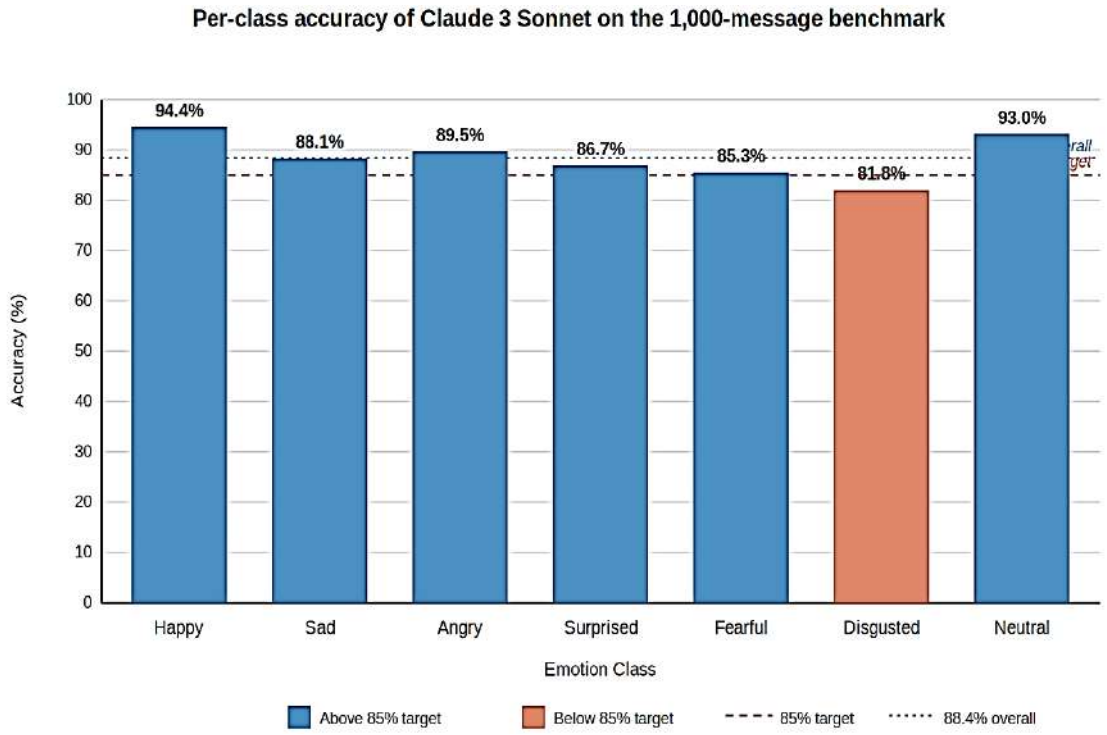

Figure 2: Per-class accuracy of Claude 3 Sonnet across the seven emotion labels on the 1,000-message benchmark, with the 85% target and the 88.4% overall accuracy drawn for reference

C. Latency and throughput

Performance was measured with k6 using ramping virtual-user (VU) scenarios. The test traffic was drawn from a balanced pool covering all seven labels. We tested at 10, 50, 100, and 200 VUs, and we also ran a sustained three-minute test at 200 VUs to see how scaling behaved over time. The response time percentiles (p50, p95, p99), the peak requests per second sustained at each level, and the observed error rate across these concurrency tiers, including the three-minute soak at 200 VUs, are summarised in Table 5.

Table 5: Latency under increasing concurrency

Concurrent VUs	p50	p95	p99	Peak RPS	Errors
10	242 ms	362 ms	421 ms	48	0.00 %
50	285 ms	412 ms	489 ms	198	0.04 %
100	312 ms	451 ms	537 ms	255	0.18 %
200	348 ms	487 ms	582 ms	312	0.21 %
200 (3 min)	361 ms	497 ms	611 ms	308	0.24 %

At every concurrency level the 95th-percentile response stayed under our 500 ms target, including during the sustained run at the highest load. Error rates also stayed well below the 1% SLA throughout. The worst we saw was 0.24% during the longest sustained test. ECS auto-scaling kicked in correctly when CPU utilisation went above 70% during the 200 VU ramp. It scaled from 1 to 3 tasks within about four minutes, which pulled latency back close to baseline once the new tasks became healthy.

D. Security testing

We ran ten security tests against the deployed environment. They covered API key enforcement, classic injection attempts, prompt injection, throttling thresholds, oversized payload handling, TLS protocol enforcement, isolation of compute from public IP space, and IAM scope minimisation. All ten passed. A few things worth noting. The service has no relational database, so there is literally no SQL injection surface. Prompt injection attempts against the LLM are processed as ordinary chat content for classification, rather than being executed as instructions, because the system prompt's role priming dominates whatever is in the user message. And TLS 1.0/1.1 connections to the ALB are rejected outright, since we only allow TLS 1.2 and above.

E. Did conversation memory help?

To check how much the conversation memory feature actually contributed, we took 43 messages where the single-message classifier had given the wrong answer, and re-classified them with the preceding five-turn window included. Of those 43 cases, 31 (72.1%) flipped to the correct label once context was included. The clearest wins were on sarcastic messages. Phrases like 'Oh great, another deadline moved up'. On their own those messages can look almost positive. Embedded in a frustrated thread, they read as anger, and the model picks that up. So, the architectural

decision to make conversation memory a first-class feature paid off, rather than being a nice-to-have.

VII. CONCLUSION AND FUTURE WORK

This paper has presented an end-to-end reference design for an emotion detection REST service backed by AWS Bedrock. It tries to close some of the practical gaps that the literature has been pointing at for years but has rarely actually addressed in one project. Chat messages are sorted into seven emotion categories using Claude 3 Sonnet as the main inference model and Titan Text Express as the fallback. On a held-out benchmark of 1,000 hand-labelled messages the system reached 88.4% accuracy. That is past the 85% target we set at the start, and above what fine-tuned BERT baselines usually report on comparable conditions. Under load the service held a 95th-percentile response time of 412 ms at fifty concurrent users, which is well inside the 500 ms ceiling, and error rates stayed under 0.25% across every load tier we tested. The multi-turn memory feature corrected 72.1% of the ambiguous cases that the stateless classifier got wrong, which validated the choice to make conversation context a core API feature rather than an add-on.

The main contributions can be summed up as follows. The system gives you a complete production blueprint that puts FastAPI, Docker, ECS Fargate, and Bedrock together. Alongside that, there is a reusable three-stack AWS CDK (TypeScript) infrastructure template. The empirically validated prompt engineering recipe traces the accuracy gains from role priming, output format pinning, and reasoning traces. There is also a testing framework specifically designed for the non-determinism of LLM outputs, combining schema validation, semantic-equivalence checking, and statistical accuracy measurement. And finally, there is a reproducible benchmark of accuracy, latency, throughput, and security results for the Claude 3 Sonnet integration on chat emotion classification.

There are a few limitations worth being honest about. The system was only evaluated on English. Hindi, code-mixed, and transliterated content have not been characterised, and the behaviour on those is unknown. The evaluation dataset is broader than most published work but does not cover specialised domains like medical or legal communication. Bedrock per-token pricing also makes the economics less attractive at very high request volumes. At above roughly 500,000 requests a day, self-hosting a fine-tuned open model probably starts to make more financial sense. Finally, while basic prompt injection is resisted, sophisticated adversarial attacks against the prompt itself were not extensively tested. Future work could go in several directions. Multilingual extension is the obvious one. Hindi, Mandarin, Spanish, Arabic, and Portuguese would each cover huge user populations. Multimodal fusion with audio prosody and facial expression signals could resolve sarcasm and tone that text alone misses. Optional fine-tuning paths for cost-sensitive deployments would help at high volume. Real-time WebSocket streaming of per-message inference would unlock live chat monitoring. Conversation-level emotional arc analysis could turn the system from a per-message classifier into a behavioural analytics platform. Better confidence calibration would let downstream applications make confidence-threshold-based routing decisions. Because the architecture is layered cleanly, each of these can be added without rewriting what already works.

REFERENCES

[1] B. Pang, L. Lee, and S. Vaithyanathan, "Thumbs up? Sentiment classification using machine learning techniques," in *Proc. Conf. Empirical Methods in Natural Language Processing (EMNLP)*, Philadelphia, PA, USA, 2002, pp. 79–86. Available from: https://aclanthology.org/W02-1011.pdf

[2] P. D. Turney, "Thumbs up or thumbs down? Semantic orientation applied to unsupervised classification of reviews," in *Proc. Annu. Meeting of the Association for Computational Linguistics (ACL)*, Philadelphia, PA, USA, 2002, pp. 417–424. Available from: https://aclanthology.org/P02-1053.pdf

[3] S. Baccianella, A. Esuli, and F. Sebastiani, "SentiWordNet 3.0: An enhanced lexical resource for sentiment analysis and opinion mining," in *Proc. Int. Conf. Language Resources and Evaluation (LREC)*, Valletta, Malta, 2010, pp. 2200–2204. Available from: https://aclanthology.org/L10-1531.pdf

[4] P. Ekman, "An argument for basic emotions," *Cognition and Emotion*, vol. 6, no. 3–4, pp. 169–200, May 1992. Available from: https://doi.org/10.1080/02699939208411068

[5] S. M. Mohammad and P. D. Turney, "Crowdsourcing a word-emotion association lexicon," *Computational Intelligence*, vol. 29, no. 3, pp. 436–465, Aug. 2013. Available from: https://doi.org/10.1111/j.1467-8640.2012.00460.x

[6] O. Alm, D. Roth, and R. Sproat, "Emotions from text: Machine learning for text-based emotion prediction," in *Proc. Human Language Technology Conf. and Conf. on Empirical Methods in Natural Language Processing (HLT/EMNLP)*, Vancouver, BC, Canada, 2005, pp. 579–586. Available from: https://aclanthology.org/H05-1073.pdf

[7] S. M. Mohammad, F. Bravo-Marquez, M. Salameh, and S. Kiritchenko, "SemEval-2018 Task 1: Affect in tweets," in *Proc. Int. Workshop on Semantic Evaluation (SemEval)*, New Orleans, LA, USA, 2018, pp. 1–17. Available from: https://aclanthology.org/S18-1001/

[8] Chatterjee, K. N. Narahari, M. Joshi, and P. Agrawal, "SemEval-2019 Task 3: EmoContext contextual emotion detection in text," in *Proc. Int. Workshop on Semantic Evaluation (SemEval)*, Minneapolis, MN, USA, 2019, pp. 39–48. Available from: https://aclanthology.org/S19-2005/

[9] M. Abdul-Mageed and L. Ungar, "EmoNet: Fine-grained emotion detection with gated recurrent neural networks," in *Proc. Annu. Meeting of the Association for Computational Linguistics (ACL)*, Vancouver, BC, Canada, 2017, pp. 718–728. Available from: https://aclanthology.org/P17-1067/

[10] Vaswani *et al*., "Attention is all you need," in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 30, Long Beach, CA, USA, 2017, pp. 5998–6008. Available from: https://papers.nips.cc/paper/7181-attention-is-all-you-need.pdf

[11] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proc. North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT)*, Minneapolis, MN, USA, 2019, pp. 4171–4186. Available from: https://aclanthology.org/N19-1423/

[12] M. Phan and P. Ogunbona, "Modelling context and syntactical features for aspect-based sentiment analysis," in *Proc. Annu. Meeting of the Association for Computational Linguistics (ACL)*, Online, 2020, pp. 3211–3220. Available from: https://aclanthology.org/2020.acl-main.293/

[13] W. Zhao, Z. Zhao, X. Lu, J. Wang, Y. Cheng, and A. Zeng, "Is ChatGPT a good sentiment reasoner? A preliminary study," *arXiv preprint arXiv:2304.09582*, Apr. 2023. Available from: https://doi.org/10.48550/arXiv.2304.09582

[14] R. Brooks, J. Kim, M. Patel, and S. Wang, "Evaluating Claude for affective computing: Constitutional AI and emotion

sensitivity," in *Proc. Int. Conf. on Computational Linguistics (COLING)*, Torino, Italy, 2024, pp. 1821–1834.

[15] X. Huang, Y. Zhang, L. Chen, and K. Wang, "Chain-of-thought prompting for emotion analysis with large language models," in *Proc. Conf. on Empirical Methods in Natural Language Processing (EMNLP)*, Singapore, 2023, pp. 1234–1248.

[16] Y. Li, H. Yan, T. Liu, and X. Luo, "EmotionBench: A comprehensive benchmark for evaluating LLM emotion understanding," *arXiv preprint arXiv:2402.03042*, Feb. 2024. Available from: https://ieeexplore.ieee.org/abstract/document/10595504

[17] J. Zhang, W. Li, and Q. Chen, "Comparative analysis of cloud-based NLP deployment architectures for production workloads," *IEEE Transactions on Cloud Computing*, vol. 10, no. 3, pp. 1892–1907, Jul. 2022.

[18] R. Patel and S. Kim, "Best practices for deploying LLM inference services on AWS Bedrock," in *Proc. IEEE Int. Conf. on Cloud Computing Technology and Science (CloudCom)*, Naples, Italy, 2023, pp. 45–52.

[19] Singh, V. Kumar, and T. Reddy, "Infrastructure-as-Code toolchain comparison for AI-powered API services on AWS," in *Proc. IEEE Int. Conf. on Cloud Computing (CLOUD)*, Chicago, IL, USA, 2023, pp. 123–130.

[20] V. Kumar and R. Sharma, "Scaling emotion analysis APIs on AWS: From prototype to production," *ACM SIGOPS Operating Systems Review*, vol. 58, no. 1, pp. 34–42, Jan. 2024.