GenAI (for) Systems

Adit Kolli, Anup Bagali, Aryan Joshi, Keshav Singh

Key idea: Improve AI results through engineering instead of optimizing models

"We define a Compound AI System as a system that tackles AI tasks using multiple interacting components, including multiple calls to models, retrievers, or external tools."

Examples:

Alphacode 2 - generates 1 million possible solution for a task and then filters

Alpha geometry - combines LLM with symbolic solver to solve olympiad problems

RAG - aids LLM with documents

COT32 inference strategy to query 32 times in Google Gemini

People moving from using one LLM call to complex inference strategies

Why build compound systems

- Better returns vs cost of building a compound system instead of just increasing model size

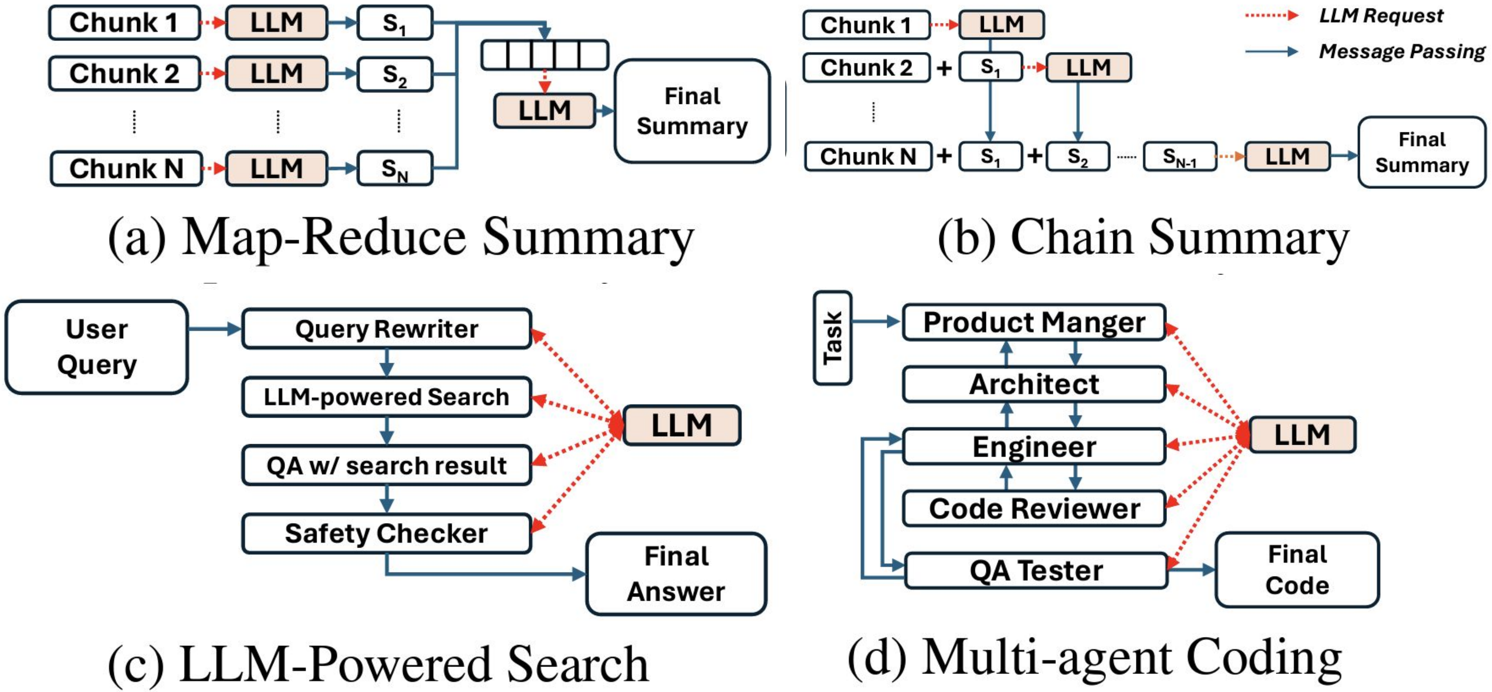
- Iterating on systems is faster

- Control and trust is easier to create with systems

Parrot: Efficient Serving of LLM Based Applications with Semantic Variable

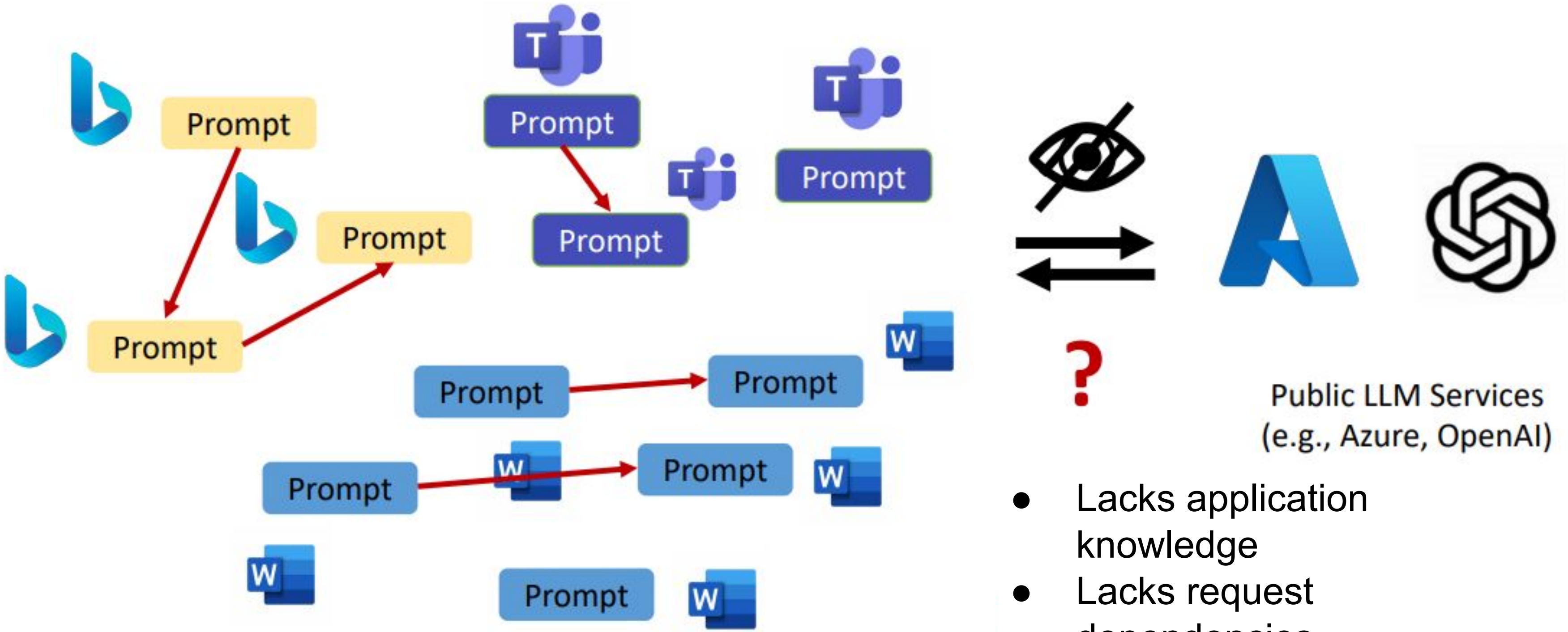
Chaofan Lin, Zhenhua Han, Chengruidong Zhang, Yuqing Yang, and Fan Yang, Chen Chen, Lili Qiu,

LLM Backends Service Diverse Applications



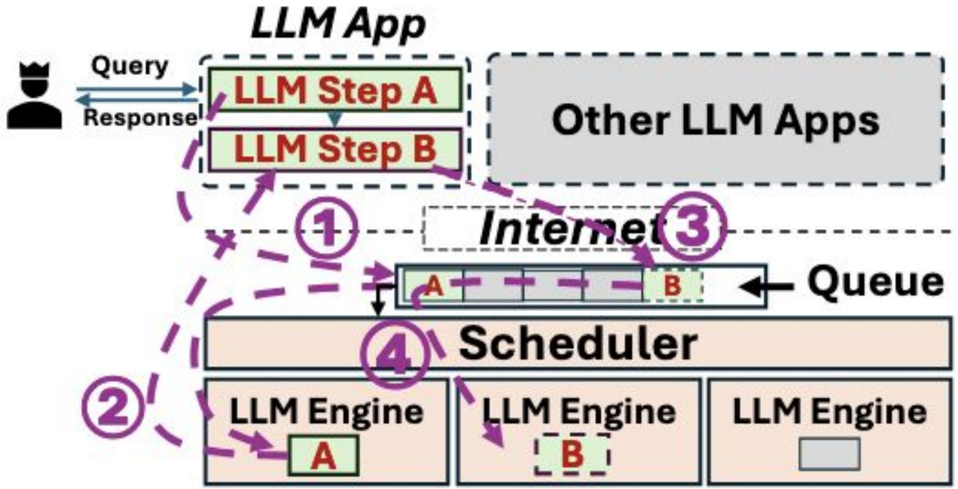
- All use the same LLM backend

LLM Services are Context-Unaware



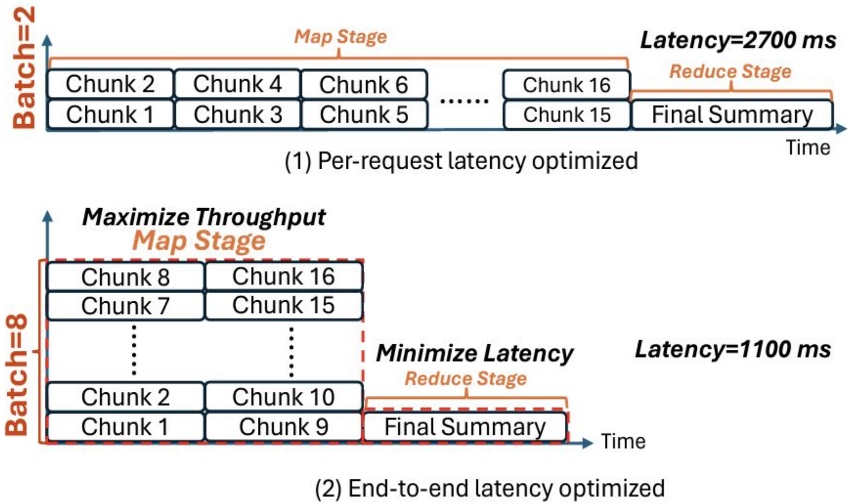
- Lacks application knowledge
- Lacks request dependencies

Problem #1: Lacking Application Knowledge Adds Latency



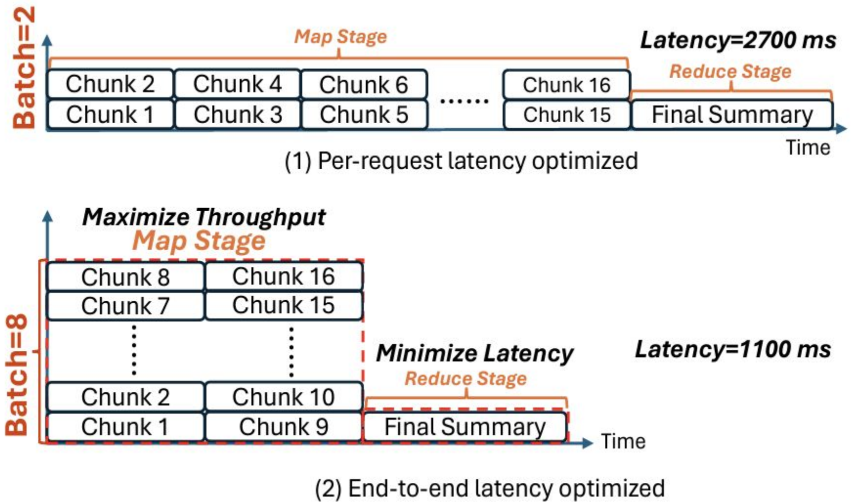
- Defaults to regular “chat” serving
- High Request Latency
  - Unnecessary Rounds Trips
  - GPU Underutilization
  - Excessive queueing delays

Problem #2: Misaligned Scheduling Objectives

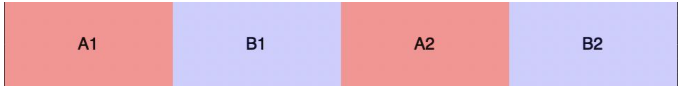


- End-to-End optimization vs. Single request optimization
  - Optimizing single requests not always optimal for end-to-end requests
  - High batch size delays single requests but optimizes end-to-end requests

Problem #2: Misaligned Scheduling Objectives



- End-to-End optimization vs. Single request optimization
  - Optimizing single requests not always optimal for end-to-end requests
  - High batch size delays single requests but optimizes end-to-end requests
- Interleaving single requests from different end-to-end requests can be suboptimal



Interleaved Execution: A=3s, B=4s

Non-Interleaved Execution: A=2s, B=4s

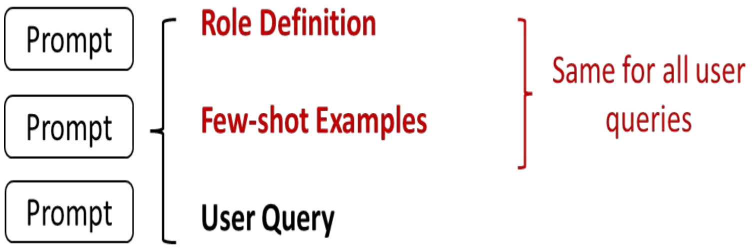
Problem #3: Inability to Exploit Correlated Requests

- Applications use similar prefixes/prompts
- No knowledge of shared structure
  - Independent requests receive “final” prompt
  - Cannot easily detect shared structure between requests

LLM-based App.	# Calls	Tokens	Repeated (%)*
Long Doc. Analytics	$2 \sim 40$	$3.5k \sim 80k$	3%
Chat Search	$2 \sim 10$	$5k$	94%
MetaGPT [22]	14	$17k$	72%
AutoGen [54]	17	$57k$	99%

*We count a paragraph as repeated if it appears in at least two LLM requests.

Table 1: Statistics of LLM calls of LLM applications.

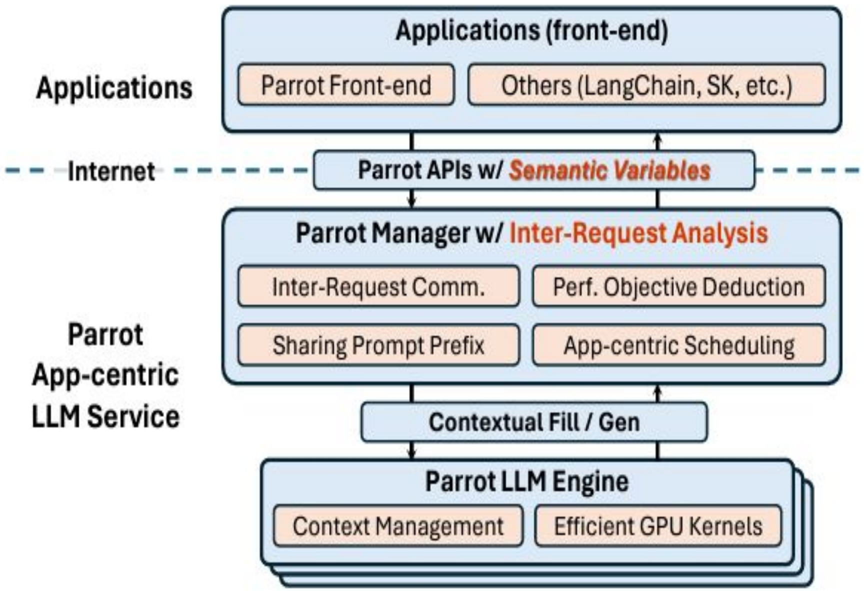


No
Application
Context

$\longrightarrow$

Blindly apply
optimizations for
all use cases

Parrot System Overview

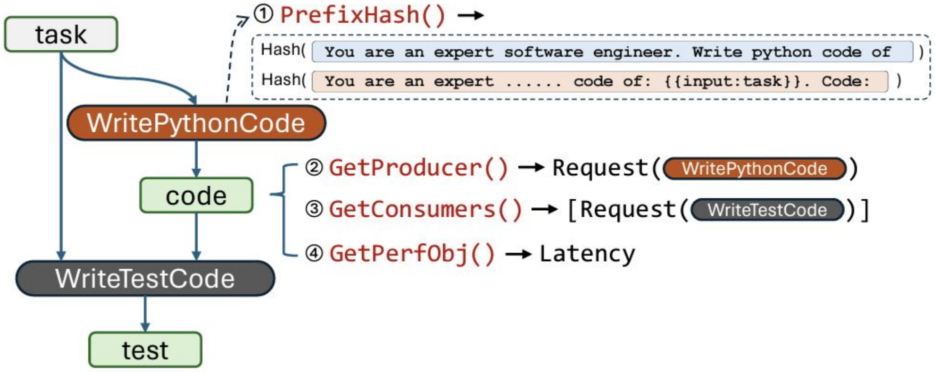


- Parrot Front-End: Allows application information to be exposed to LLM Service
- Parrot Manager: Cluster Level Scheduler that utilizes application information for optimized scheduling
- Parrot LLM Engine: Efficient engines that exploit prefix sharing

Parrot Front-End: Semantic Variables

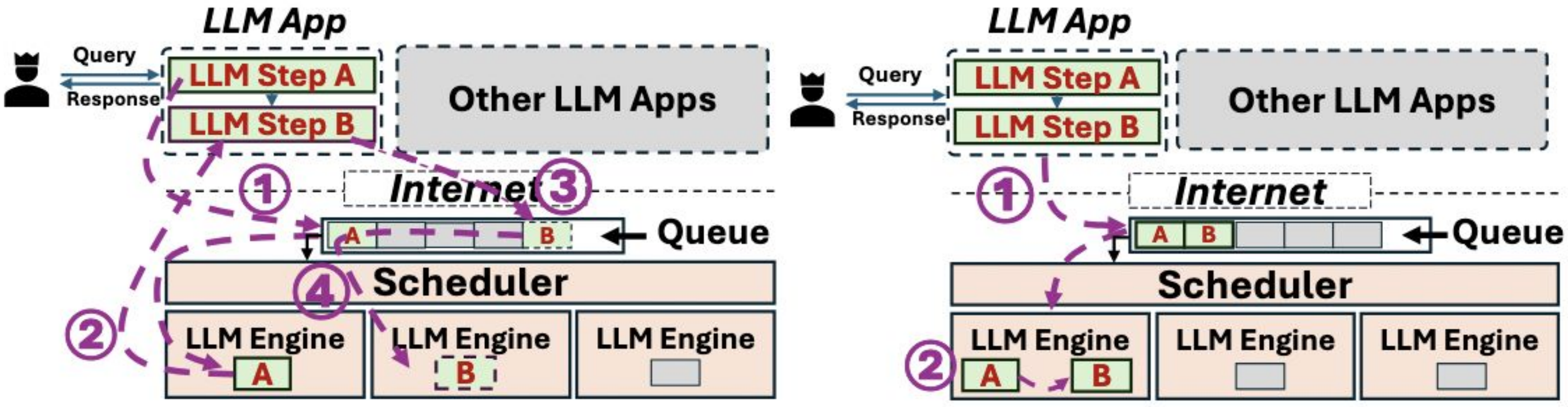
```python
@P.SemanticFunction
def WritePythonCode(task: P.SemanticVariable):
""" You are an expert software engineer.
    Write python code of {{input:task}}.
    Code: {{output:code}}
"""

@P.SemanticFunction
def WriteTestCode(
    task: P.SemanticVariable,
    code: P.SemanticVariable):
""" You are an experienced QA engineer.
    You write test code for {{input:task}}.
    Code: {{input:code}}.
    Your test code: {{output:test}}
"""

def WriteSnakeGame():
  task = P.SemanticVariable("a snake game")
  code = WritePythonCode(task)
  test = WriteTestCode(task, code)
  return code.get(perf=LATENCY), test.get(perf=LATENCY)
```

- Inter-request “data pipes”
- Semantic functions represents requests, variables their input/output
- Used to construct dependency DAG



Parrot Front-End: Less Round Trips Reduces Latency

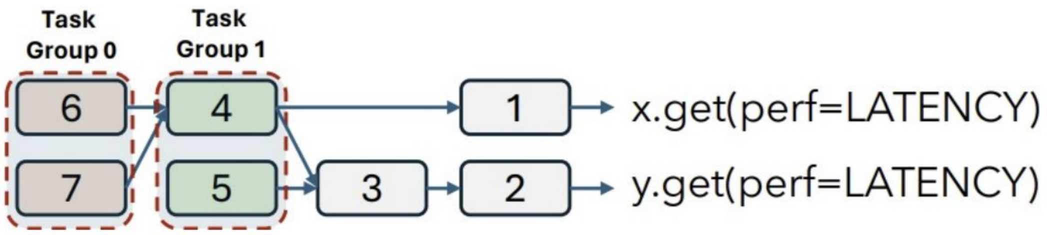
- Only single round trip between Parrot Front-End and Back-End
- Reduction in network delay from reduced interactions
- Less queuing delays by reducing communication overhead



(b) Current LLM Services (c) Our system: Parrot

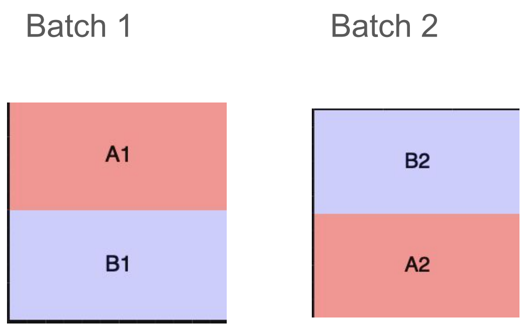
Parrot Manager: DAG-Based Analysis

- DAG-based analysis identifies dependencies between requests
- Requests scheduled after all input semantic variables available
- Requests at same topological level scheduled together in task group
- Application-level performance criteria determines scheduling of task groups

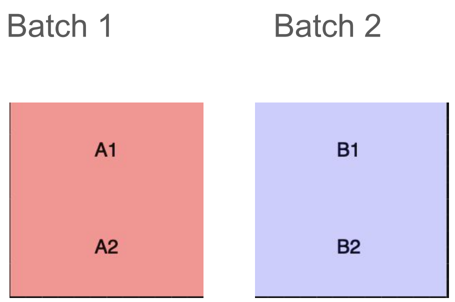


Parrot Manager: Application-Aware Scheduling

- Parrot prioritizes the following when scheduling requests:
    1. Scheduling requests from same application together to avoid interleavings



FIFO Scheduling

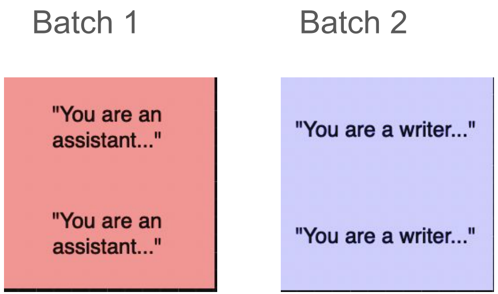


Application-Wise Grouping

Parrot Manager: Application-Aware Scheduling

- Parrot prioritizes the following when scheduling requests:
  1. Scheduling requests from same application together to avoid interleavings
  2. Maximize sharing by scheduling requests at machine with prefix

FIFO Scheduling



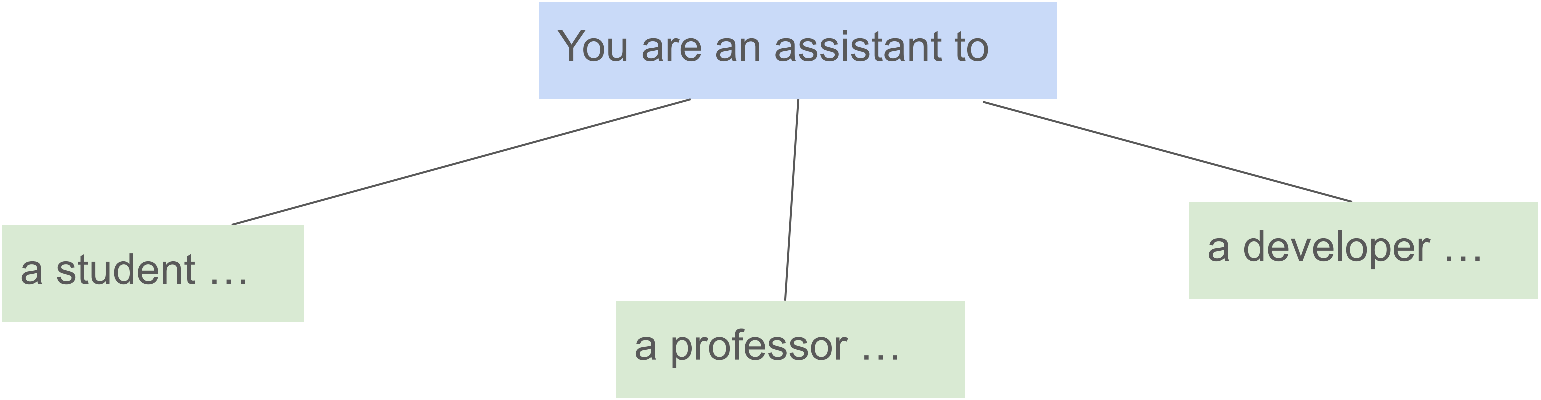
Prefix-wise Grouping

Parrot Manager: Application-Aware Scheduling

- Parrot prioritizes the following when scheduling requests:
    1. Scheduling requests from same application together to avoid interleavings
    2. Maximize sharing by scheduling requests at machine with prefix
    3. Find best engine for request given performance criteria: lower capacity machines for latency sensitive requests

Parrot Engine: Shared Prompt Prefix Optimizations

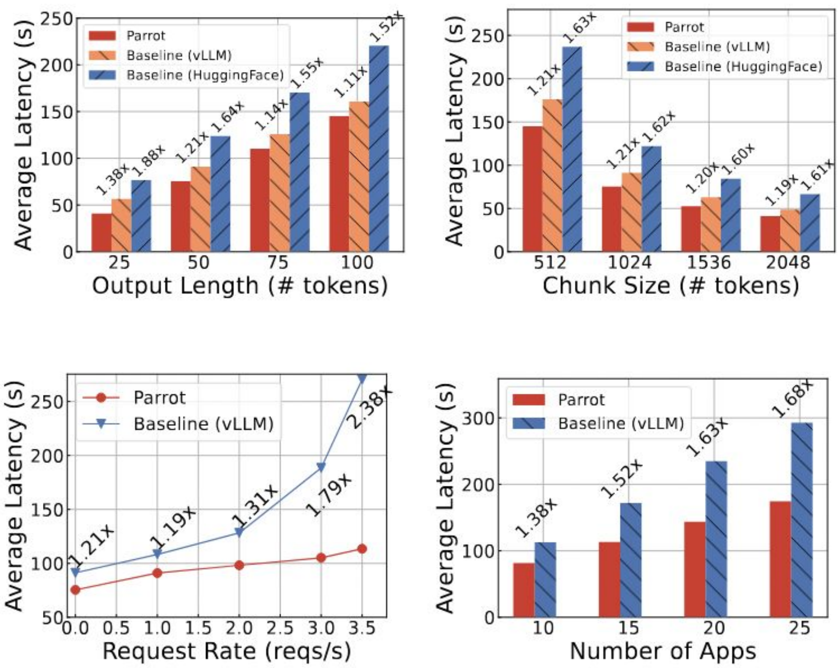
- KV cache tiles for shared prefix loaded only once to shared memory
- Interim attention values for prefix computed once and stored in HBM
- Partial attention for suffix amalgamated with prefix results for final output



Experimental Setup

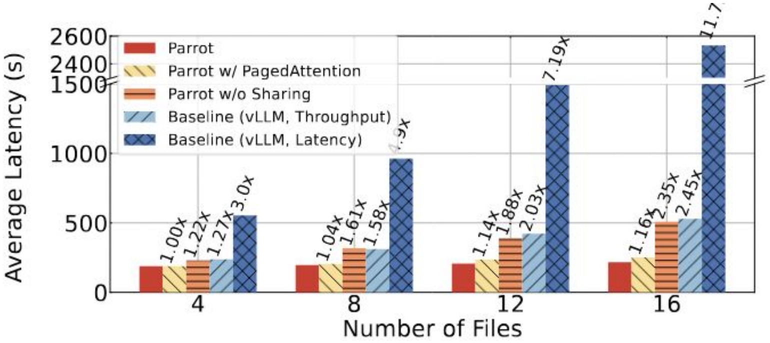
- Profiled Parrot performance under a variety of workloads
  - LLaMA 13B or LLaMA 7B
- Single-GPU and Multi-GPU setups
  - 1 A100 GPU
  - 4 A6000 GPUs
- Compared E2E latency & throughput against SOTA serving frameworks
  - vLLM, HuggingFace Transformers

Experiment: Chain Summarization (i.e. Data Analytics)

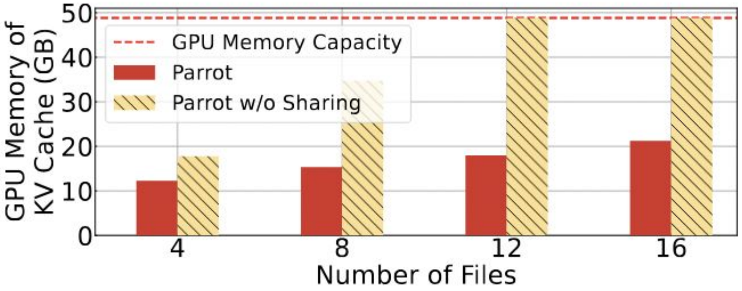


- Subsequent summaries rely on previous summaries
- Compared end-to-end latency of chain summarization tasks against vLLM and HuggingFace
- Single Application vs. Multiple Concurrent Applications
- Parrot consistently has lower latency

Experiment: Multi-Agent Coding



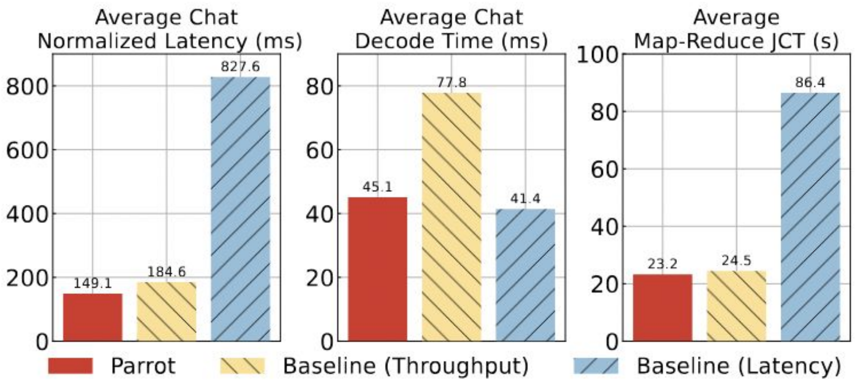
(a) End-to-end Latency



(b) GPU Memory of KV Cache

- Multiple LLM requests assume "Architect", "Coder", "Tester" roles to develop a program
- Significant prompt sharing and dependencies
- Compared E2E latency + effect of shared prefix kernel
- Parrot reduces latency and conserves GPU memory w/ custom kernel

Experiment: Mixed Workloads



- Concurrent Map-Reduce summarization and Chat workloads
- Compared latency for Parrot E2E optimization vs. throughput/latency optimization
- Parrot significantly reduces E2E latency and Job Completion Time (JCT)

Limitations and Future Work

- Dynamic Applications and Function Calling: Parrot doesn’t support dynamic control flow and native functions
- Other Applications of Inter-Request Analysis: Application level information can be used to implement other scheduling features (e.g fairness, delay scheduling)
- Integrating Parrot into LLM Orchestration Frameworks: Existing Frameworks can incorporate Parrot LLM Service for optimizing end-to-end requests

Automatic Root Cause Analysis via Large Language Models for Cloud Incidents

Yinfang Chen, Huaibing Xie, Minghua Ma, Yu Kang, Xin Gao, Liu Shi, Yunjie Cao
Xuedong Gao, Hao Fan, Ming Wen, Jun Zeng, Supriyo Ghosh, Xuchao Zhang
Chaoyun Zhang, Qingwei Lin, Saravan Rajmohan, Dongmei Zhang, Tianyin Xu
Microsoft

Background

What is root cause analysis for cloud incidents?

Identifying the root of an issue/alert that arose in a production setting.

Example: An alarm that we set up automatically detects that users are not able to log on to our email service**(alert)**.

We find that this is because we misconfigured the certificates **(root cause)**

Traditionally performed by **on call engineers**

They look at data sources like **logs, traces, metrics**

Troubleshooting Guide (TSG) - documentation that helps on call engineers identify the issue and apply mitigation steps

Problems

- On call can be **time consuming and confusing** for engineers when incidents arise
- **Volume of data can be overwhelming** for engineers and makes it difficult to identify root cause of a problem quickly using manual inspection
- TSG might not fit specific issue or be out of date
- Hard to determine **root cause of new problems**

Potential solution: Automation of RCA data collection and analysis

Key Observations

Insight 1: Determining the root cause based on a single data source can be challenging

Insight 2: Incidents stemming from similar or identical root causes often recur within a short period

Insight 3: Incidents with new root causes occur frequently and pose a greater challenge to analyze

RCA Copilot output

Provides **categorization** and **rationale** of why that category was the prediction

The prediction of "I/O Bottleneck" was made based on the occurrence of System.IO.IOExceptions within crucial functions handling input/output operations, suggesting an issue with data processing. The nested exception within the DiagnosticsLog module reinforces this notion. These errors, combined with crashes on different backend machines, point to a system struggle with handling data flow.

RCA Copilot Overview (10,000 foot view)

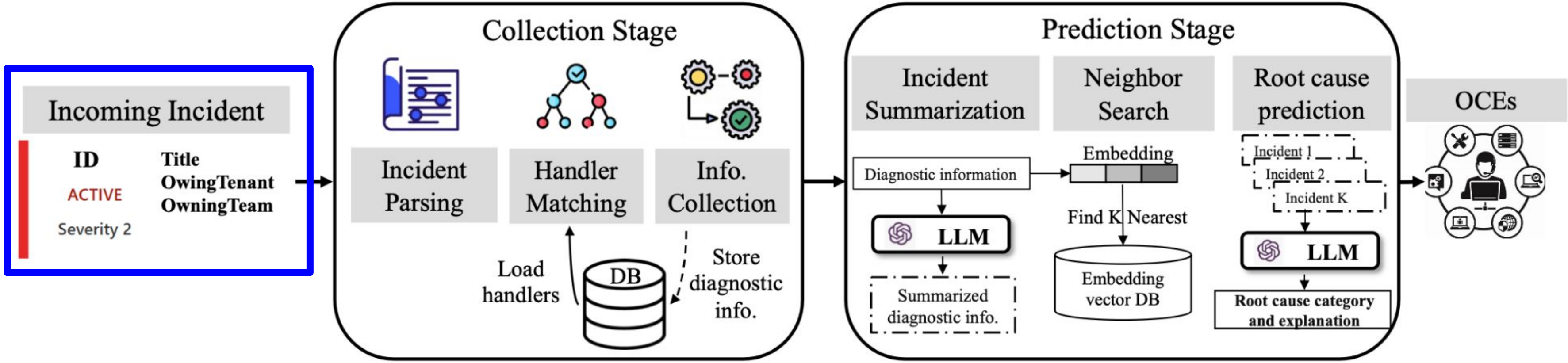


Figure 4. RCACopilot architecture.

1) **Incoming incident -** Some alert is raised upon incident, (i.e. metric anomaly)

RCA Copilot Overview (10,000 foot view)

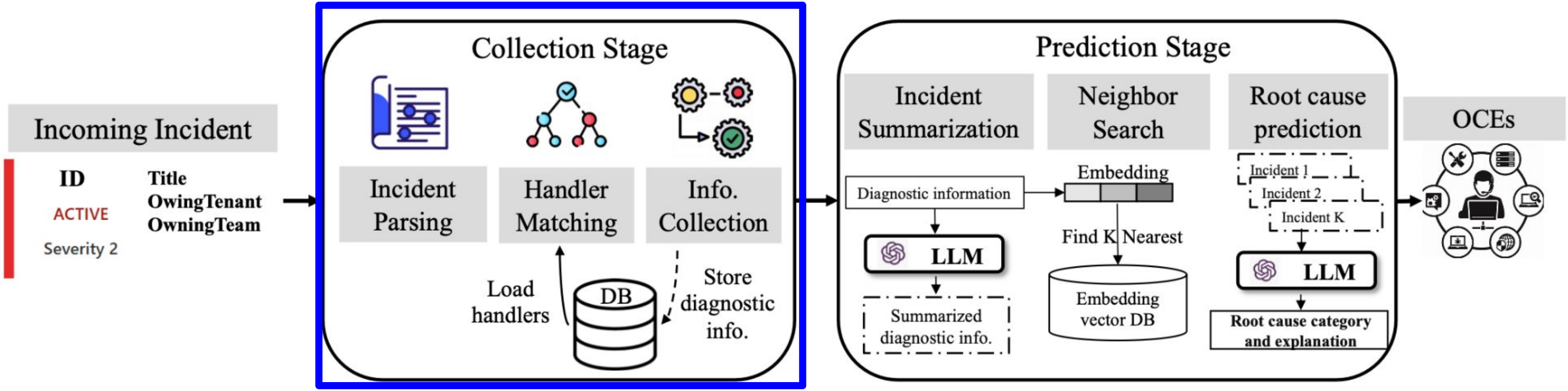


Figure 4. RCACopilot architecture.

1) **Incoming incident -** Some alert is raised upon incident, (i.e. metric anomaly)
2) **Collection Stage -** Information collection process is automated by Engineer defined workflows

RCA Copilot Overview (10,000 foot view)

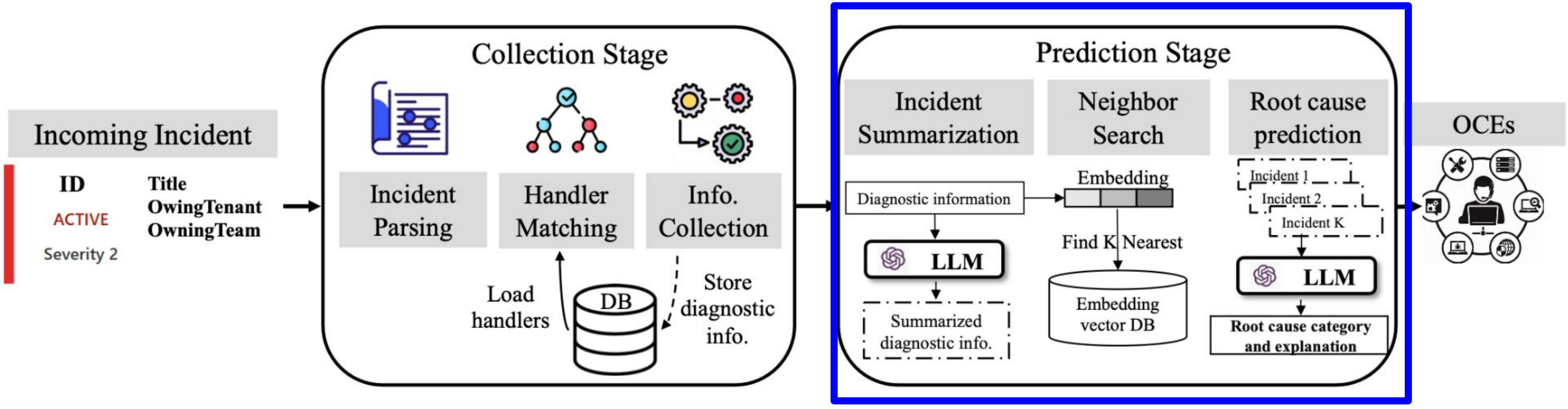


Figure 4. RCACopilot architecture.

1) **Incoming incident -** Some alert is raised upon incident, (i.e. metric anomaly)
2) **Collection Stage -** Information collection process is automated by Engineer defined workflows
3) **Prediction Stage -** Diagnostic information is passed to LLM which uses past incident information to provide root cause prediction and rationale behind prediction

Collection stage overview

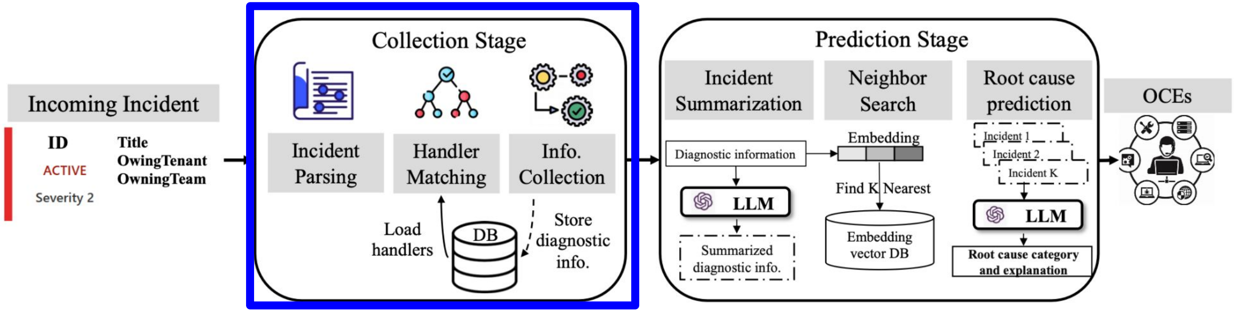


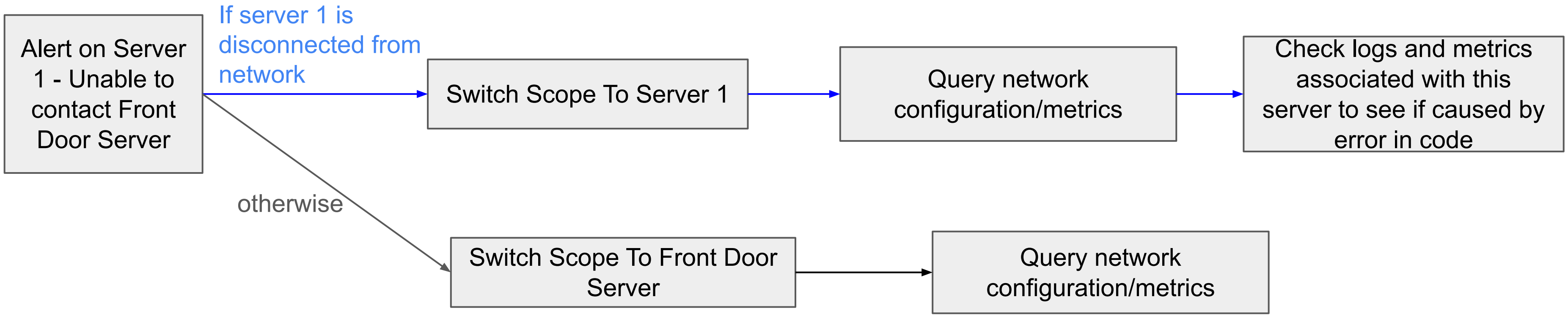
Figure 4. RCACopilot architecture.

Begins when monitor reports incident which gets matched with predefined workflow

Incident handlers: Engineers build workflow from components called handler actions. Actions can query logs/metrics/traces from different machines. Engineers can construct a decision tree of handler actions where decisions are based on output of actions.

Output: All relevant metrics and logs

Collection Stage example



Scenario: Your team manages an email server to send emails to external domains

Investigation: Monitor alerts Server 1 failure to connect to front door server

Workflow results in query to Front Door Server’s network metrics such as number of UDP ports in use

Collected information will be passed into Prediction Stage

Prediction Stage Overview

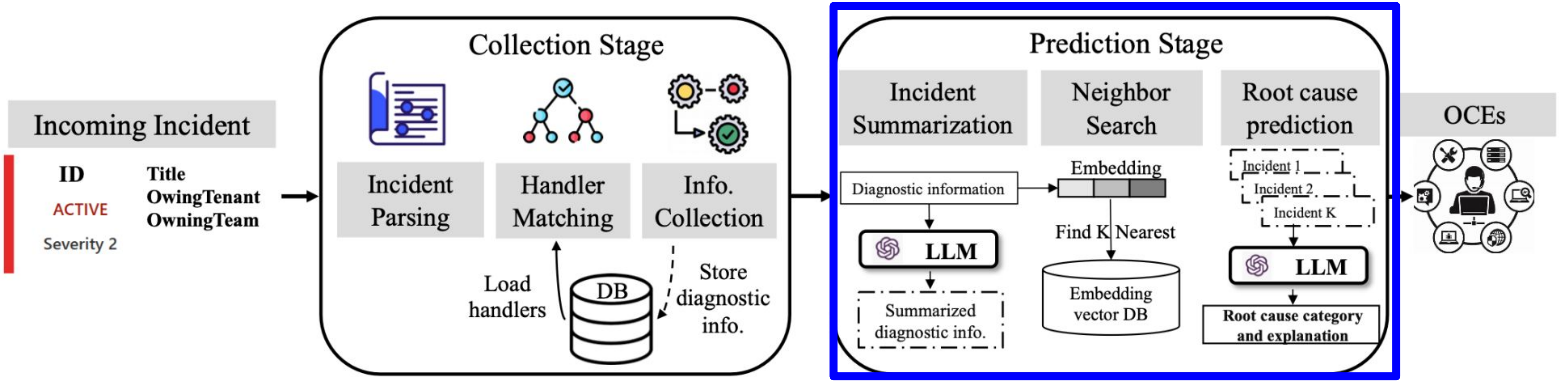


Figure 4. RCACopilot architecture.

Input: All diagnostic information collected in collection stage

1. **LLM summarizes diagnostic information** to extract key information since LLMs have trouble analyzing large text
2. **Nearest neighbor search performed** to find similar incidents
3. **Nearest neighbor examples and summary provided to LLM** for category prediction

Prediction Stage: Summarization

- Diagnostic information and prompt provided to LLM to get summary of diagnostic information

```
DatacenterHubOutboundProxyProbe probe log result from
[MachineID].
Total Probes: 2, Failed Probes: 2
 Id  Level  Created                 Description
 -   --     ---                     ----
 2   Error  11/21/2022  2:04:20 AM  Probe result
 2   Error  11/21/2022  1:49:20 AM  Probe result
Failed probe error:
Name: No such host is known.
A WinSock error: 11001 encountered when connecting to
host: [HOST NAME]
Count: 2
. . .
Exceptions:
InformativeSocketException: No such host is known.
A WinSock error: 11001 encountered when connecting to
host: [HOST NAME]
at TcpClientFactory.Create(...)
at SimpleSmtpClient.Connect(...)
. . .
Total UDP socket count: 15276
Total UDP socket count by process and processId (top
5 only):
14923: Transport.exe, 203736
15: w3wp.exe, 102296
8: svchost.exe, 4748
7: Microsoft.Transport.Store.Worker.exe, 74060
7: Microsoft.Transport.Store.Worker.exe, 87724
```

"The DatacenterHubOutboundProxyProbe has failed twice on the backend machine, with both failures due to a WinSock error 11001 indicating that the host is unknown. This error was encountered while attempting to connect to the host. The error is associated with the EOP service and has not been notified yet. The failure context suggests the same issue. ***The total UDP socket count is 15276, with the majority being used by the Transport.exe process.*** *The issue seems to be related to the SMTP connection and requires further investigation."*

"Please summarize the above input. Please note that the above input is incident diagnostic information. The summary results should be about 120 words, no more than 140 words, and should cover important information as much as possible. Just return the summary without any additional output."

Prediction Stage: Nearest Neighbor Search

- **K nearest neighbor search** is performed to find similar incidents
- Nearest neighbor search uses embedding similarity and time similarity because of **Insight 2 - Incidents stemming from similar or identical root causes often recur within a short period**
- **Formula intuition -** α determines weight given to temporal difference; higher α = more emphasis on time, less emphasis on embedding distance

$$Distance(a, b) = ||a - b||_2$$

$$Similarity(a, b) = \frac{1}{1 + Distance(a, b)} * e^{-\alpha|T(a) - T(b)|}$$

Prediction Stage: Nearest Neighbor Search

- **K nearest neighbor search** is performed to find similar incidents
- Nearest neighbor search uses embedding similarity and time similarity because of **Insight 2 - Incidents stemming from similar or identical root causes often recur within a short period**
- Unsummarized diagnostic info is **embedded** using FastText trained on historical incidents
- **Vector DB stores** embeddings of previously seen incidents along with corresponding summary and category
- **Formula intuition -** α determines weight given to temporal difference; higher α = more emphasis on time, less emphasis on embedding distance

$$Distance(a, b) = ||a - b||_2$$

$$Similarity(a, b) = \frac{1}{1 + Distance(a, b)} * e^{-\alpha|T(a) - T(b)|}$$

Prediction Stage: Embedding

- Unsummarized diagnostic info is **embedded** using FastText trained on historical incidents
- **Vector DB stores** embeddings of previously seen incidents along with corresponding summary and category
- Vector DB allows us to **efficiently find similar incidents** to the one we are diagnosing and retrieve their categorization and summary

Prediction Stage: Root Cause Prediction

- **Few shot CoT prompt** constructed using nearest neighbor summary and category prediction
- Used to get category prediction and reasoning for why the LLM made that prediction

Context: The following description shows the error log information of an incident. Please select the incident information that is most likely to have the same root cause and **give your explanation** (just give one answer). If not, please select the first item "Unseen incident".
Input: The DatacenterHubOutboundProxyProbe probe result from [BackEndMachine] is a failure ...
Options:
A: Unseen incident.
B: The DatacenterHubOutboundProxyProbe has failed twice ... *category:* ***HubPortExhaustion***.
C: There are 62 managed threads in process TransportDelivery ... *category:* ***AuthCertIssue***.

Few shot CoT prompt example

The prediction of "I/O Bottleneck" was made based on the occurrence of System.IO.IOExceptions within crucial functions handling input/output operations, suggesting an issue with data processing. The nested exception within the DiagnosticsLog module reinforces this notion. These errors, combined with crashes on different backend machines, point to a system struggle with handling data flow.

Category prediction + rationale example

Experiments: Alternative Architectures

- Used the dataset from Microsoft email service to assess efficacy of various models when directly fed diagnostic data
- Evaluated using **F1-Score,** harmonic mean of precision and recall and **inference and training time**
- **FastText** - lightweight text embedding model used in other RCA studies
- **XGBoost** - parallel tree boosting model
- **GPT3.5 Fine Tuned** - Tuned with training set, without COT
- **GPT-4 Prompt** - RCACopilot without few shot examples
- **GPT-4 Embed** - RCACopilot that uses GPT embedding instead of FastText embedding

Table 2. Effectiveness of different methods.

Method	F1-score		Avg. Time (s)	
	Micro	Macro	Train.	Infer.
FastText [61]	0.076	0.004	10.592	0.524
XGBoost [3]	0.022	0.009	11.581	1.211
Fine-tune GPT [1]	0.103	0.144	3192	4.262
GPT-4 Prompt	0.026	0.004	–	3.251
GPT-4 Embed.	0.257	0.122	1925	3.522
RCACopilot (GPT-3.5)	0.761	0.505	10.562	4.221
RCACopilot (GPT-4)	**0.766**	**0.533**	10.562	4.205

Experiments: Input Data

- RCACopilot's approach of providing **only summarized diagnostic info** to LLM produces best results

- Demonstrates that excess information hurts LLM accuracy

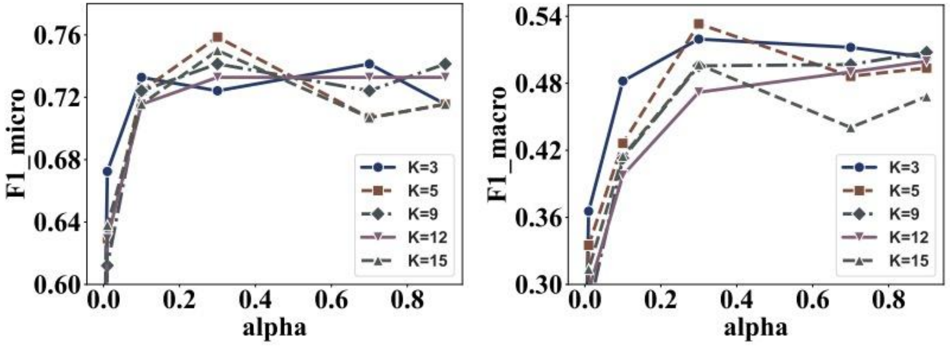
Data Source			F1-score	
AlertInfo	DiagnosticInfo	ActionOutput	Micro	Macro
	✓		0.689	0.510
	✓$^{\text{sum.}}$		**0.766**	**0.533**
✓			0.379	0.245
✓	✓		0.525	0.511
✓		✓	0.431	0.247
	✓	✓	0.501	0.449
✓	✓	✓	0.440	0.349

AlertInfo: Initial data provided by monitor

ActionOutput: output of handlers

Experiments: Prompting

- Evaluating best **K** and **alpha**
- **K** is the number of examples provided in each few shot prompt
- **Alpha** is the weight given to temporal distance. **Higher alpha means time has more emphasis, embedding distance has less emphasis**
- Best results came at **K = 5, alpha = 0.3**



(a) F1 micro. **(b)** F1 macro.

Deployment Results

- Data collection without category prediction deployed by 30 teams for four years
- Average times reported by teams to do data collection is significant improvement over manual methods

Team	Avg. exec. time (seconds)	# Enabled handler
Team 1	841	213
Team 2	378	204
Team 3	106	88
Team 4	449	42
Team 5	136	41
Team 6	91	34
Team 7	449	32
Team 8	255	32
Team 9	323	31
Team 10	22	18

Limitations

- RCACopilot only activated when incident handler is defined for monitor alert **(Some monitor alerts do not have handlers)**
- Some incidents can not be detected by monitors
- Each incident handler must be manually built by engineers (**Not fully automated)**
- Leaves RCACopilot susceptible to mistakes that engineers make when defining handler workflows

Thank you for listening!

Parrot Appendix: Workload Optimizations Summary

Workload	Serving Dependent Requests.	Perf. Obj. Deduction	Sharing Prompt	App-centric Scheduling
Data Analytics	✓	✓		✓
Serving Popular LLM Applications			✓	✓
Multi-agent App.	✓	✓	✓	✓
Mixed Workloads	✓	✓		✓