SNIA DEVELOPER CONFERENCE
SDC 25
By Developers FOR Developers

Hyatt Regency Santa Clara, CA
September 15-17, 2025

Enhancing Defect Triaging in Storage Systems using AI from Integration Test Knowledge Graphs

Nuthan Prasad B N, Dhishankar Sengupta, Manoj Srivatsav

HPE

Contents

Abstract

Bug detection and triaging in complex storage systems present challenges unlike those in general-purpose or SaaS software. Storage stacks must interact simultaneously with the OS kernel, device drivers, and hardware, adding layers of complexity in logging, concurrency, and control flow.

Hundreds of threads and processes frequently write to shared logs without transactional guarantees, creating noisy, interleaved traces that conventional AI bug trackers—trained mostly on generic code—struggle to interpret.

Key Innovation: We propose augmenting system code with knowledge extracted from high-level integration test cases to create contextual understanding for AI-driven bug triaging.

Integration tests capture realistic, end-to-end behaviors that unit tests miss. We transform their scenario intent and observed effects into a structured knowledge graph modeling components, interactions, inter-process communications, and hardware events.

Problem

Multi-layered Complexity

Complex multi-layered failures across hardware, firmware, and software stacks with massive concurrency create debugging nightmares

Log Correlation Chaos

Log correlation challenges with interleaved, non-transactional outputs where critical failures are buried in noise

Manual Analysis Overhead

80% of debugging time spent on manual log analysis instead of actual problem resolution

Domain Expertise Gap

Existing AI tools lack domain expertise, forcing reliance on manual processes and tribal knowledge. High expertise barrier limits team scalability as only senior engineers can effectively diagnose issues.

Current Limitations

Knowledge Quality Dependency

The system's effectiveness relies heavily on accurate initial workflow extraction from test cases, with errors potentially cascading throughout the system.

Log Pattern Evolution and Drift

As software evolves, log formats and error patterns change, potentially making historical patterns outdated and reducing system accuracy over time.

Context Window Limitations

Large-scale failures generating extensive logs may exceed LLM context windows, leading to incomplete analysis and missed critical information.

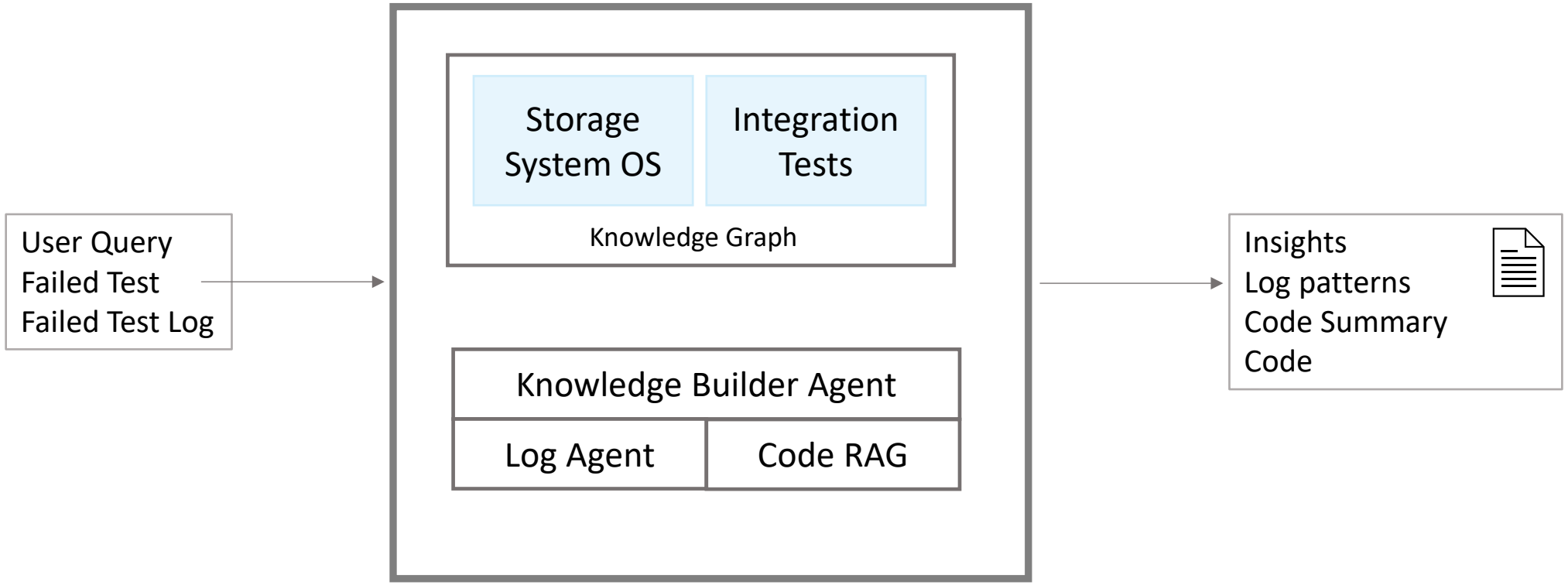
→ **False Positive Predictions**

The predictive system may flag tests as potentially failing when they would actually pass, causing unnecessary alarm fatigue.

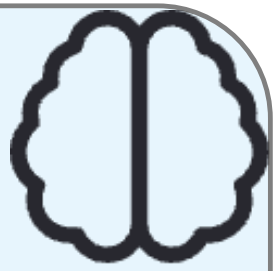
→ **Novel Failure Pattern Handling**

The system struggles with entirely new types of failures or edge cases not seen in training data.

Solution Overview



Technical Infrastructure



- Hosted LLM Model - Private deployment using transformers library
- Open API Compatible Server - LLM Model interaction layer
- Embedding Models - Semantic understanding of code and logs
- Vector Database - High-performance storage for embeddings

Code Parsing Layer

- AST-based Chunking - Intelligent code parsing and segmentation
- Semantic Code Splitting - Context-aware code decomposition
- Code RAG System - Retrieval-Augmented Generation for code context
- Make model understand customer-specific or domain-specific data:

Log Analysis & Behavior Mapping

Function Flow Capture Process

- Track complete call graph of every function invoked during test execution
- Monitor parameter tracking for input/output data flow between function calls
- Record system state before and after each function execution
- Document code path coverage showing specific branches and conditions executed

Test Case Intelligence Extraction

- Extract scenario validation to map what specific business scenario each test validates
- Perform function coverage analysis to identify partial versus complete function exercising
- Create dependency mapping to track external systems and resources accessed
- Conduct performance profiling for execution time and resource usage per function

Log Analysis & Behavior Mapping (Cont'd)

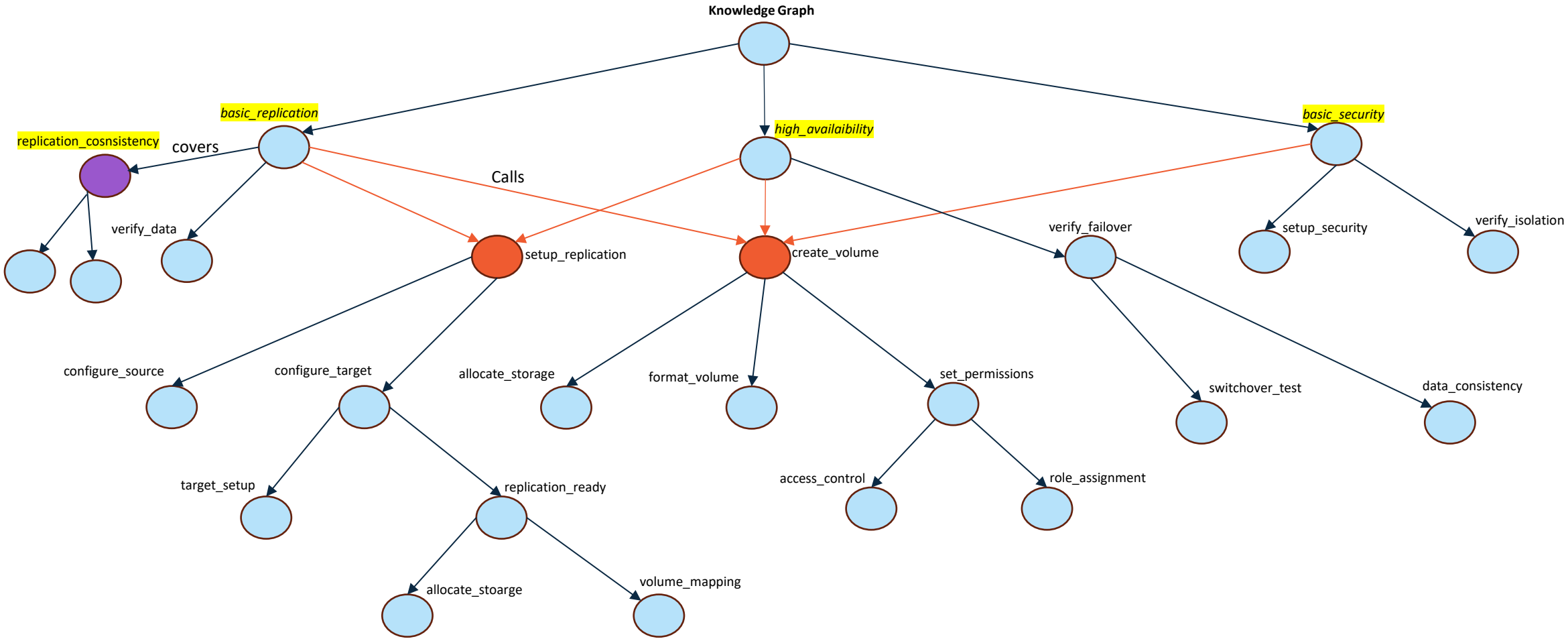
Multi-Test Function Profile Building

- Build comprehensive function profiles through incremental knowledge construction
- Capture test coverage data showing which test cases exercise each function
- Map code paths linking happy path, error handling, and performance scenarios to specific tests
- Record execution profiles for normal, under load, and error scenario conditions

Cross-Test Intelligence Building

- Combine multiple tests to create complete function profiles
- Add different execution paths and error conditions from various test scenarios
- Establish performance baselines from various load conditions
- Build comprehensive library of expected versus actual execution patterns

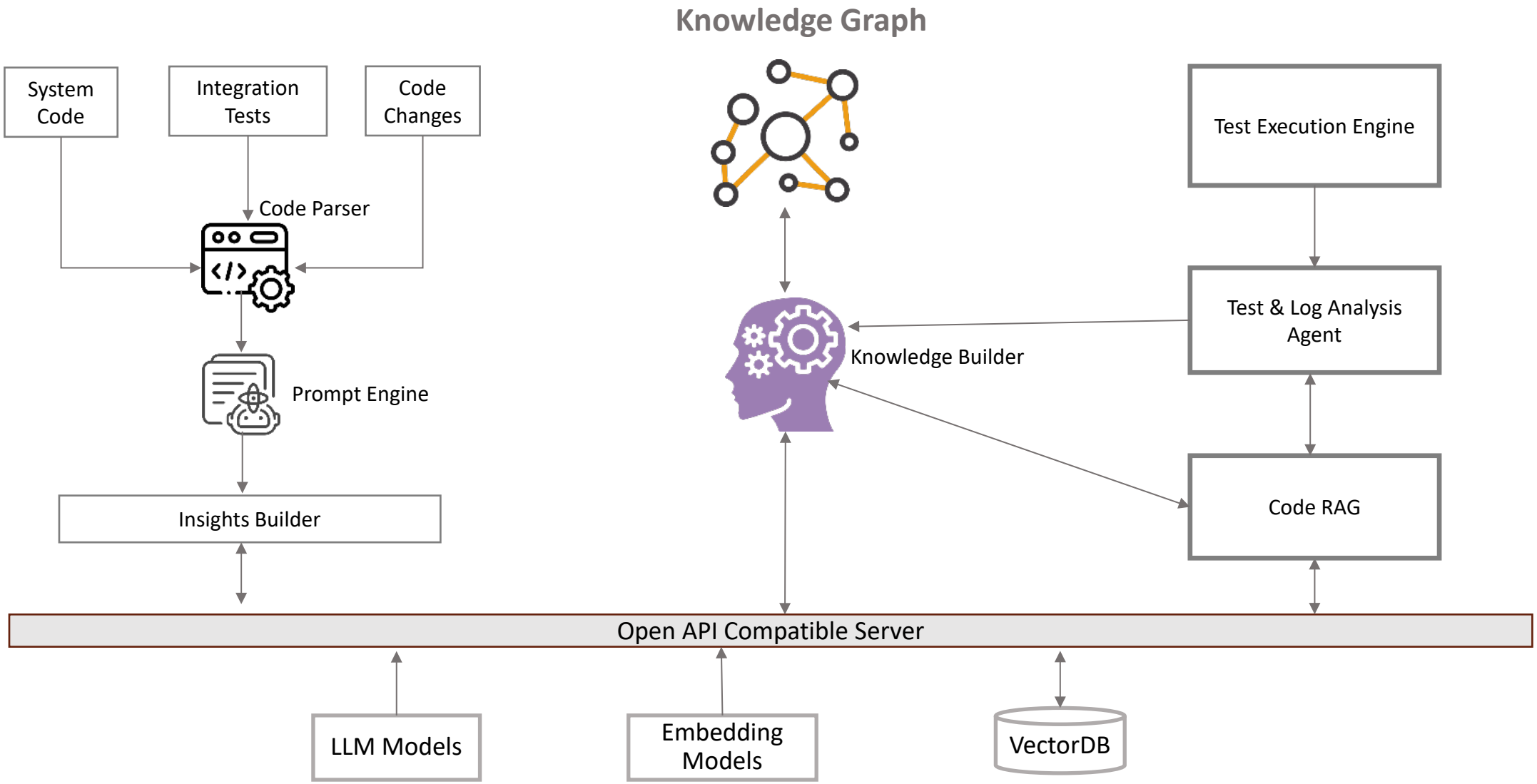
Knowledge Graph



Knowledge Data Structure


```
{
  "function_name": "allocate_storage",
  "file_path": "/src/storage/allocation/storage_allocator.py",
  "function_summary": "Allocates storage space on secondary array for replication setup",
  "keyphrases": ["storage allocation", "secondary array", "space allocation", "replication storage"],
  "code_block": {
    "start_line": 45,
    "end_line": 65,
    "code": "def allocate_storage(volume_id, size_gb, secondary_array_ip):\n    logger.info(f'Starting storage
  },
  "scenarios": {
    "happy_path": {
      "description": "Normal storage allocation process",
      "log_patterns": [
        "Starting storage allocation on secondary array",
        "Connecting to secondary array at {ip_address}",
        "Requesting {size}GB storage allocation",
        "Storage allocation completed for volume {volume_id}"
      ]
    },
    "storage_service_timeout": {
      "description": "Secondary array storage service timeout",
      "log_patterns": [
        "Starting storage allocation on secondary array",
        "Connecting to secondary array at {ip_address}",
        "Requesting {size}GB storage allocation",
        "Storage service response delayed",
        "Storage allocation request timeout after 30s"
      ],
      "failure_indicators": ["storage_timeout", "service_unresponsive", "allocation_failed"]
    }
  }
}
```


Architecture



Example

Step 1: Test Fails with Generic Message

FAILED test_high_availability_failover – Replication setup failed

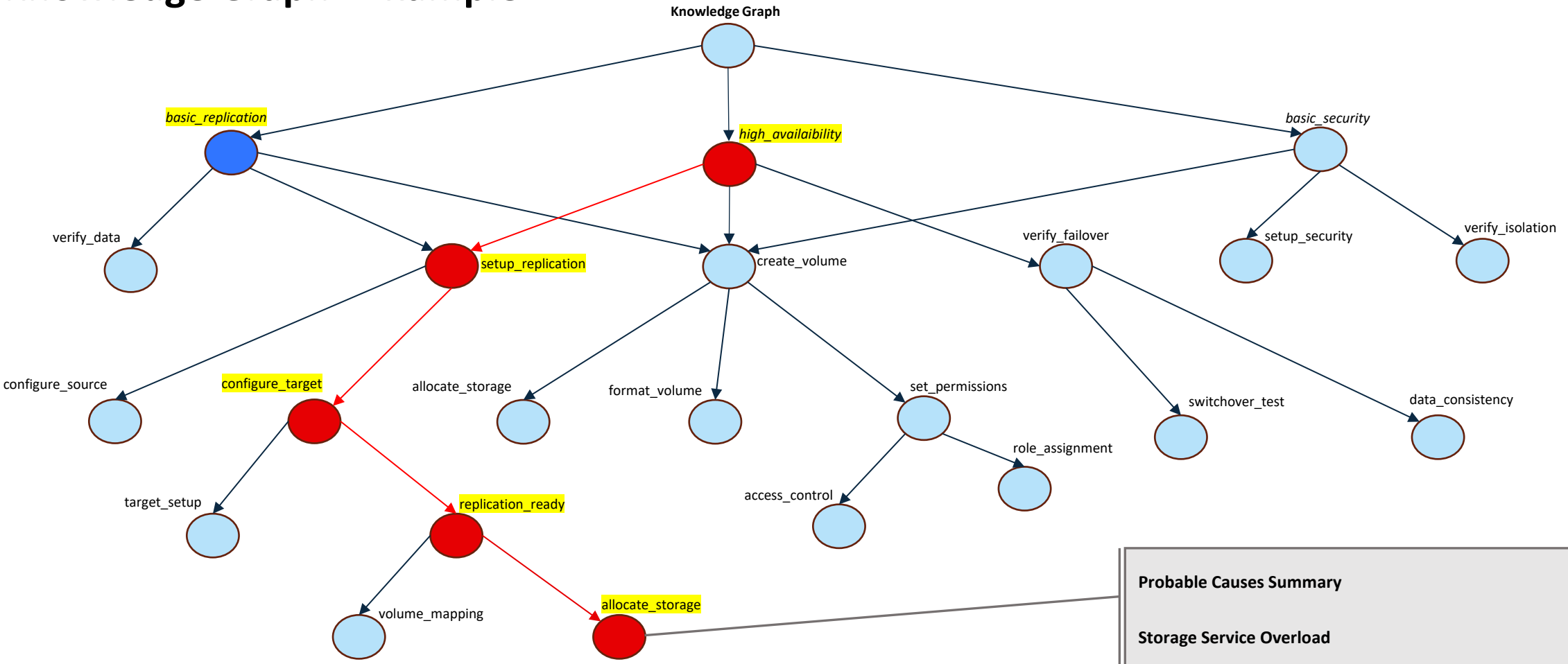
Step 2: Logs

```
2025-09-14 14:23:15.123] INFO  test_high_availability_failover - Starting high availability test
[2025-09-14 14:23:15.245] INFO  create_volume - Volume vol_12345 created successfully on primary array
[2025-09-14 14:23:15.367] INFO  setup_replication - Initiating replication setup for volume vol_12345
[2025-09-14 14:23:15.489] INFO  replication_ready - Checking replication readiness
[2025-09-14 14:23:15.612] INFO  allocate_storage - Starting storage allocation on secondary array
[2025-09-14 14:23:15.734] INFO  allocate_storage - Connecting to secondary array at 10.50.12.200
[2025-09-14 14:23:15.856] INFO  allocate_storage - Requesting 500GB storage allocation
[2025-09-14 14:23:25.234] WARN  allocate_storage - Storage service response delayed
[2025-09-14 14:23:35.567] WARN  allocate_storage - Still waiting for storage allocation response
[2025-09-14 14:23:45.890] ERROR replication_ready - Storage allocation failed - aborting replication setup
[2025-09-14 14:23:45.991] ERROR setup_replication - Replication setup failed.
[2025-09-14 14:23:46.012] FAIL  test_high_availability_failover - AssertionError: Replication setup failed
```

Step 3: Agentic approach - Queries Pre-Built Knowledge Graph

- Looks up the test workflow setup_replication() → replication_ready() → allocate_storage()
- Retrieves the function profile for allocate_storage()

Knowledge Graph - Example



Failure Pattern

- **Entry Point:** Line 45 in /src/storage/allocation/storage_allocator.c
- **Failure Mode:** Response timeout after successful request
- **Duration:** 30 seconds (exceeded threshold)
- **Exit Code:** Storage allocation failed

Probable Causes Summary

Storage Service Overload

```
# Line 45-50 in allocate_storage()
response = secondary_array_client.allocate_space(
volume_size=requested_size,
timeout=30 # ← TIMEOUT HERE
)
```


Benefits

Dramatic Time Savings

Significant reduction in defect triaging time from hours to minutes through automated root cause analysis.

Enhanced Accuracy & Prevention

High accuracy in failure analysis with precise pinpointing of locations and proactive detection of potential issues before they occur.

Operational Excellence

Faster issue resolution leading to reduced production incidents, accelerated release cycles, and higher software reliability.

Conclusion

Revolutionary Quality Assurance

Our AI-powered solution transforms manual debugging into an intelligent, automated system that understands code behavior and predicts failure patterns.

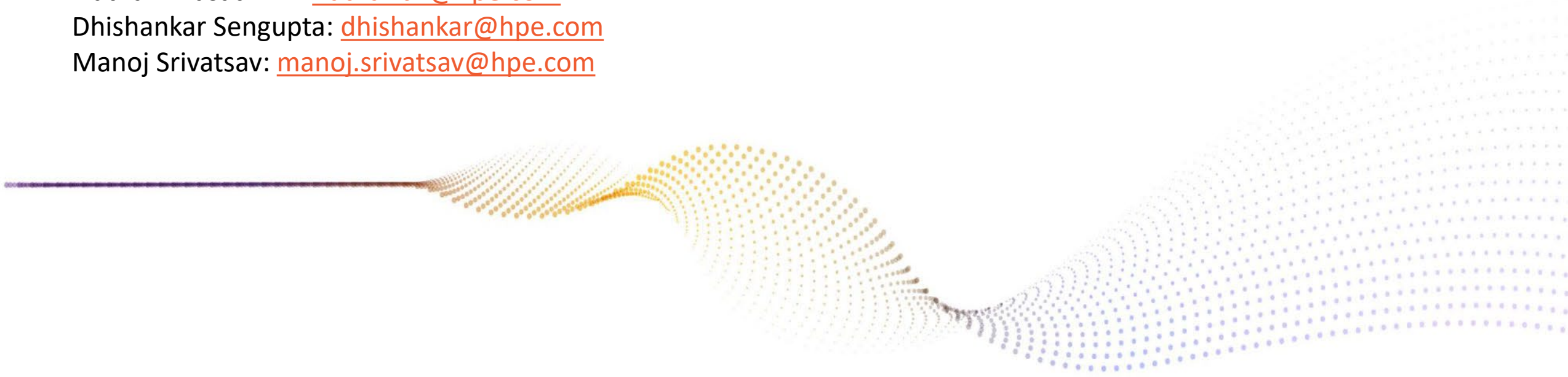
Immediate Strategic Impact

Delivers substantial productivity gains while establishing the foundation for next-generation AI-driven development tools, positioning our team as leaders in intelligent software engineering.

Future Enhancement Roadmap

Opens pathways for automated test generation, intelligent code suggestions, predictive optimization, and self-healing system recommendations.

Nuthan Prasad B N: nuthanbn@hpe.com
Dhishankar Sengupta: dhishankar@hpe.com
Manoj Srivatsav: manoj.srivatsav@hpe.com



Thank you for attending!

Please remember to rate this session. You get access the presentations at
http://sniadeveloper.org/conference

