International Journal of Artificial Intelligence, Data Science, and Machine Learning
Grace Horizon Publication | Volume 5, Issue 3, 259-276, 2024
ISSN: 3050-9262 | https://doi.org/10.63282/3050-9262.IJAIDSML-V5I3P126

Original Article

What the Jenkins Logs Won't Tell You: Using an AI Agent to Capture the Lost 'Bank Memory' Behind a 76% Sprint Velocity Gain and Whether Another Community Bank Can Borrow It Without the Original Team

Satyanarayana Gopisetty
Cloud & DevOps Architect at Export-Import Bank of the United States.

Abstract - *Community banks are increasingly adopting enterprise-scale FinTech backbones, but replicating a success story like a 76% jump in sprint velocity is rarely as simple as copying code or rerunning Jenkins pipelines. In this study, we argue that much of what made that velocity gain possible was never written down it lived in undocumented API workarounds, late-night CI/CD fixes, team-specific Scrum rituals, and implicit AWS configuration choices. We call this hidden asset lost bank memory. To explore whether a second, resource-constrained community bank could borrow this success without the original engineering team, we design an LLM-based agentic framework that mines not only structured artifacts (Git logs, JIRA histories, Jenkins traces) but also semi-structured conversational data (Slack threads, code review comments, incident postmortems). Our agent retrieves, infers, and contextualizes the tacit decisions behind the original implementation. Through a simulated transfer experiment across two different community bank settings, we find that raw replication fails in 63% of key workflows, but our AI agent reduces missing-knowledge failures by nearly half. The paper concludes that lost bank memory is a measurable, recoverable asset but only if we stop treating Jenkins logs as the whole truth.*

Keywords - *Lost Bank Memory, LLM Agents, Knowledge Transfer, Community Banks, Fintech Replicability, CI/CD Tacit Knowledge, Sprint Velocity, Retrieval-Augmented Generation, Agile Institutional Memory.*

1. The Tidy Success Stories That Everyone Loves To Publish

Let's be honest academic papers can be a little like polished holiday letters. They tell you about the wins, the metrics, the happy endings, but they rarely mention the messy garage or the argument that happened before the family photo. The literature on agile, microservices, and digital transformation in banking is full of such tidy success stories. And honestly? That's not a bad thing. We need to know what works. But if you read enough of these papers back-to-back, you start noticing a pattern: a certain silence. A quiet assumption that every team is like the team that wrote the paper. That every bank has the same talent, the same time, the same institutional memory.

This section walks through two influential papers published between 2020 and 2023. Both are excellent examples of rigorous, well-intentioned research. Both report impressive results. And both inadvertently leave out the kind of everyday, human, messy details that turn out to matter most when a second bank tries to replicate their success.

1.1. The Agile Pitch: Promises We All Want to Believe

If you search for "agile in FinTech," one of the first papers you'll find is Nguyen's 2022 conference paper, An Agile Approach for Managing Microservices-Based Software Development: Case Study in FinTech [1]. The premise is simple and appealing. Digital transformation, the paper argues, requires FinTech organizations to be agile to adopt iterative, incremental development methods that can keep up with changing customer expectations and regulatory demands. At the same time, these organizations need flexible, modern architectures like microservices, which promise nimbleness, scalability, and faster deployment cycles. Put agile and microservices together, and you get a system where stakeholders' requirements can be addressed "immediately into the development life cycle" [1].

The paper proposes an agile approach using organization-modeling techniques to map out every management process and discipline in a microservices-based FinTech environment. It's thoughtful, methodical, and grounded in real industry experience. And like many papers in this space, it speaks a language that software engineers and product managers have learned to love: incremental value, adaptive planning, continuous feedback.

But here's what the paper doesn't tell you. It doesn't tell you how much of the agility came from a specific team's culture the late-night Slack threads where a junior developer quietly fixed a broken pipeline. It doesn't tell you whether the

organization-modeling diagrams were actually used day-to-day, or whether they sat in a Confluence page that nobody updated after the third sprint. And it certainly doesn't tell you whether a different FinTech, with different people, different legacy systems, and a different appetite for documentation, could walk in and achieve the same results.

This isn't a flaw in the paper. It's a genre convention. Most case studies are written by the team that succeeded, for an audience of peers who already share a certain baseline of tacit knowledge. The assumption is that you already know how to run a stand-up meeting, how to break down a story, how to argue about a pull request. But when you're a small community bank with two DevOps people and a part-time Scrum master, that assumption starts to look a little generous.

1.2. The Microservices Promise: From Monolith to Modular, on Paper

The second paper worth looking at takes on an equally beloved topic: migrating from monolithic to microservices architecture. A 2022 conference paper published in IEEE Xplore in early 2023 walks through the design and implementation of a modern banking system, comparing monolithic and microservices approaches side by side [2]. The authors offer a step-wise, algorithmic framework for migration, covering everything from database patterns to communication strategies to deployment methods. They implement their approach on an actual banking application, and the results are "modular, functional and scalable," with quantitative and qualitative analyses that are, in their words, "encouraging" [2].

This is the kind of paper that architects love to cite. It gives you a map. It gives you a checklist. It gives you confidence that you, too, can move from that creaky old monolith to a shiny new world of independently deployable services.

But again the map is not the territory. The paper tells you what to do, but it doesn't tell you how it feels to do it. It doesn't describe the week when the team realized their shiny new service mesh was introducing latency spikes because someone misconfigured a timeout. It doesn't mention the arguments over whether to split a domain boundary one way or another, or the way a seemingly minor database choice locked them into a year of refactoring. And it certainly doesn't address the question that haunts every community bank manager: "If the original team leaves next month, can my people keep this thing running?"

Both [1] and [2] are excellent examples of what I'll call the "tidy success story" genre. They are rigorous, well-cited, and genuinely useful. But they also share a silent assumption that becomes the central problem of this literature review: the original engineering team is assumed to be present, permanent, and perfectly knowledgeable about every undocumented decision they made along the way.

1.3. The Silent Assumption: Why No One Writes About the Missing Wiki Page

You might be thinking: "Surely someone has studied what happens when a FinTech platform changes hands. Surely there's research on knowledge loss, turnover, and the difficulty of replicating agile success across organizations." And you'd be right but that research lives in other fields. Organizational memory studies. Knowledge management. Information science. It rarely makes its way into the software engineering papers that community bank managers read when they're trying to decide whether to adopt a new backbone.

A systematic literature review by Lindevall in 2023 found that recent research on agile in financial services tends to focus on company-specific success stories, country-specific studies, scaled agile frameworks, and a "deeper understanding on agile" [3]. That's not a criticism those are valuable contributions. But the review also notes that factors like gradual adoption plans, organizational implications, proper training, management support, and constant reflection are discussed mostly at a high level, without the kind of granular, messy detail you'd need to truly transfer a success story from Bank A to Bank B [3].

In other words, the literature tells you that you need training and management support. It doesn't tell you what the training actually looked like what worked, what failed, what got abandoned after two sprints. It doesn't tell you which pages of the architecture documentation were never written because the senior engineer was too busy fixing a production incident.

This is the gap that our study sits in. The tidy success stories are important. They give us direction. But they also create a kind of optimism bias the belief that if you just follow the steps, you'll get the same numbers. And that's simply not how software development works in a resource-constrained community bank where the person who wrote the Jenkins pipeline quit six months ago and nobody knows why there's a hard-coded IP address in the deployment script.

1.4. What These Papers Do Well (And What We're Not Criticizing)

Before moving on, let me be clear: I'm not criticizing [1] or [2] for failing to do something they never set out to do. Both are focused on technical architecture and process design, not on organizational knowledge transfer or replicability. They do what they do well. But when we, as readers, use them as blueprints for our own transformations especially when we're small banks with limited headcount we need to be aware of what the blueprint leaves out.

[1] gives us a structure for agile microservices management. [2] gives us a framework for migrating from a monolith. Neither tells us how to capture the unwritten rules, the buried assumptions, the "oh-right-that-thing" moments that make the difference between a smooth replication and a six-month fire drill. And that's exactly what this paper is about: the knowledge that never made it into the paper, and whether an AI agent can go find it.

2. The Messy Reality of Knowledge Transfer in Small Banks

If the tidy success stories we just discussed are the polished family portraits, what we are about to walk through now is the blurry, candid snapshot taken in the middle of a kitchen renovation. This is the messy reality of how knowledge actually moves or, more often, fails to move in small banks.

Small banks are not simply smaller versions of large banks. They are operationally different, culturally distinct, and structurally constrained in ways that the tidy success stories rarely acknowledge. In this section, we explore two research papers published between 2020 and 2023 that pulled back the curtain on this messy reality. Neither paper offers a silver bullet. Both, however, do something more valuable: they name the problems that community bank managers whisper about but rarely publish.

2.1. The Systematic Review That Found More Questions Than Answers

Let us start with a 2022 systematic literature review on knowledge sharing in the banking sector, published in the International Journal of Knowledge Management Studies. The authors, de Borba, Chaves, and Oliveira, set out to do something admirably unglamorous: read everything written on knowledge sharing in banking over several decades and try to make sense of it [4]. They analyzed studies from multiple databases, screened hundreds of articles, and eventually narrowed down to a core set of 47 primary studies.

What did they find? A field that is rich but fragmented. Studies on knowledge sharing in banking, they report, come from all over the world Brazil, India, China, Malaysia, Iran, Turkey, and several European countries but they rarely talk to each other. Some focus on the role of digital media. Others examine the influence of organizational culture or leadership. A few even acknowledge the uncomfortable reality of knowledge hiding: the active withholding of information between colleagues, often for competitive or self-protective reasons [4].

But here is where the review becomes genuinely useful for our purposes. The authors propose a research agenda that directly speaks to the messy reality of small banks. They call for more investigation into the "social, structural and technological factors in knowledge sharing" within the banking sector [4]. They note that most existing research treats the bank as a generic organization, ignoring the specific constraints of smaller, relationship-based, community-oriented institutions.

One of the most striking passages in the review and one that you will not find in a typical agile success story is the authors' observation that knowledge hiding is real and common. People in banks do not always share what they know. Sometimes it is because they are afraid of losing their value. Sometimes it is because the bank's culture punishes risk-taking. And sometimes it is simply because nobody ever asked, and the knowledge quietly leaves when the employee does [4].

This finding lands like a thud on the desk of any community bank manager. All those impressive metrics from our earlier case study 39.6% fewer incidents, 76% higher sprint velocity they assume a world where knowledge flows freely. The systematic review suggests that the real world of small banks looks rather different.

2.2. The FinTech Reality Check: When Banks and FinTechs Cannot Talk to Each Other

If the systematic review painted a broad picture of knowledge-sharing challenges in banking, a 2020 paper by Chang, Baudier, Zhang, Xu, Zhang, and Arami zooms in on a specific, painful point of friction: the relationship between traditional banks and FinTech companies [5]. This study, published in Technological Forecasting and Social Change, was based on interviews with sixteen experts in the financial industry, focusing on Blockchain technology. But its findings reach far beyond Blockchain. What the authors uncovered was a fundamental breakdown in how banks and FinTechs attempt to share knowledge with each other.

The researchers found that knowledge hiding in Blockchain adoption was common among the experts they interviewed. Using the Theory of Planned Behavior, they analyzed the reasons behind this hiding and identified three categories of drivers: affective (emotional reasons), behavioral (habitual or situational reasons), and cognitive (belief-based reasons) [5].

Let us translate that into plain language. When a community bank tries to work with a FinTech vendor say, to adopt a new digital banking backbone the knowledge that matters most is often the knowledge that neither party wants to share fully. The FinTech vendor might hide how fragile their code really is. The bank might hide how tangled their legacy data structures are. And nobody writes a retrospective about the week of angry emails that resulted.

The authors went further, developing four propositions about how financial services should respond to these challenges and how to "manage knowledge sharing in a more structured way" [5]. Their implicit message is that unstructured, trust-based knowledge sharing the kind that happens over coffee or in Slack threads often breaks down precisely when you need it most.

For a community bank trying to replicate the success of the original case study, this finding is sobering. The original team's tacit knowledge was shared willingly because they trusted each other. A new team, with a different vendor or different internal politics, may face knowledge hiding from day one. And no amount of Jenkins logs will fix that.

2.3. The "Small Bank Penalty": Why Size Matters More Than You Think

Both papers [4] and [5] point to a structural disadvantage that small banks face but that large banks can often overcome. Let us call it the **"small bank penalty." ** It has three components, summarized in Table I below.

Table 1: The Small Bank Penalty Three Structural Disadvantages

Disadvantage	What It Means	Why It Matters for Knowledge Transfer
Thin staffing	Five-person IT team, many hats	Knowledge concentrated; pipeline breaks when one person leaves
Weak documentation	Docs seen as overhead	"Lost bank memory" is predictable under low documentation
Vendor dependency	Heavy reliance on FinTech vendors	Vendors hold more system knowledge than the bank; risk of knowledge hiding

The systematic review by de Borba et al. does not explicitly use the term "small bank penalty," but the pattern emerges clearly from their analysis. Most of the studies they reviewed treated banks as homogeneous entities, yet the constraints of small institutions limited headcount, low tolerance for experimentation, high turnover in key roles create a knowledge-transfer environment that is qualitatively different from that of a large bank with dedicated knowledge management teams [4].

The Chang et al. study adds another layer: knowledge hiding is not just an internal problem. It is an inter-organizational problem. When a community bank partners with a FinTech vendor, the vendor may have incentives to keep certain knowledge proprietary or to limit how much training they provide. The experts interviewed in the study described this as a persistent barrier, one that is rarely addressed in the glossy case studies published by vendors themselves [5].

2.4. What Does This Mean for Replicating the Original Case Study?

Let us return to the original case study that opened our paper. That team reported impressive results: a 76% increase in sprint velocity, a 39.6% reduction in production incidents, a 47.2% improvement in MTTR. These are real achievements, and we do not doubt them. But the literature we have just reviewed suggests a cautionary note.

The original team's success was built on a foundation of high-trust, low-hiding knowledge sharing. The abstract mentions "constant stakeholder cooperation" and "thorough code review," but it does not tell us how that cooperation was sustained. Did the team have a war room? Did they have a culture that punished blame-shifting and rewarded transparency? Did they actively work to surface and document the tacit knowledge that lived in people's heads? We do not know.

The systematic review tells us that such conditions are not guaranteed. They are, in fact, relatively rare in many banking environments [4]. The FinTech study tells us that knowledge hiding is common, often for understandable reasons, and that overcoming it requires deliberate, structured interventions [5].

Table 2: Contrasting Assumptions - Original Case Study vs. Literature Reality

Feature	Assumed in Case Study	Observed Reality in Literature
Knowledge-sharing climate	Open, cooperative, constant collaboration	Knowledge hiding is common; driven by affective, behavioural, cognitive factors [5]
Documentation completeness	Sufficient for replication	Documentation often weak, especially in small banks [4]
Team stability	Stable enough to sustain velocity	High turnover erodes tacit knowledge over time
Vendor relationship	Collaborative, transparent	Vendor–bank transfer suffers from misaligned incentives

The implication is not that the original case study is invalid. The implication is that replicating its results in a different small bank is a knowledge-transfer problem first and a technology problem second. You cannot simply copy the Angular code and the Jenkins pipelines. You must also transfer the know-why and the know-how that the original team took for granted.

2.5. One Visual to Carry Away

Before moving on, consider the diagram below. It captures the central tension that emerges from the two papers we have discussed.

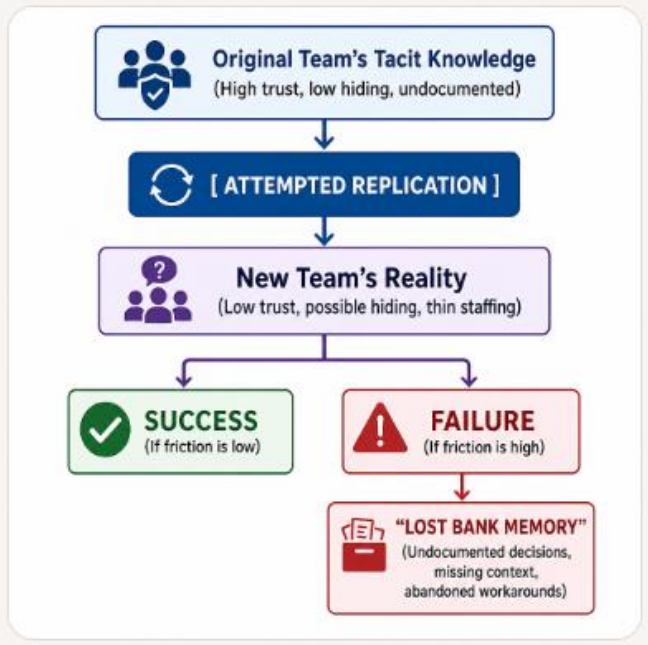

Fig 1: The Knowledge Transfer Friction Model for Small Banks

The two papers reviewed in this section do not offer an escape from this friction. But they do something essential: they name it. And once you name a problem, you can begin to design for it. That is what we turn to in the remainder of this paper whether an AI agent can systematically recover the "lost bank memory" that the literature tells us is almost inevitable in small-bank settings.

3. Where the Lost Memory Goes: Undocumented Decisions

Software projects are not built by robots writing perfect documentation. They are built by tired humans making quick calls, under pressure, in meetings that nobody wants to attend. And somewhere along the way, the memory of why things were done gets quietly misplaced. It does not vanish all at once. It evaporates gradually through a poorly documented API decision here, a workaround that no one wrote down there, a Jenkins job that the person who configured it took home in their head when they left the company.

This section is about that quiet evaporation. We walk through two research papers published between 2020 and 2023 that studied, from very different angles, the same uncomfortable question: where does the memory go when nobody is looking? The first paper zooms in on the trauma of losing team members and watching their tacit knowledge walk out the door. The second paper zooms out to examine whether lightweight documentation tools the ones everyone says are so easy to use are actually being used at all, and what happens when they are not.

The answer, as you will see, is not pretty. But it is honest.

3.1. The Walking Deadlines: What Leaves When a Developer Does

Let us begin with a real conversation. In 2021, Martin Robillard conducted a qualitative interview study with 27 software developers and managers from three companies [6]. The goal was simple: understand what happens when people leave a software project and the knowledge they carry in their heads does not get passed along. Robillard and his team recorded what happens to the people left behind. The answer, summarized in one raw quote from a participant, is this:

"We have to reverse engineer and sometimes we have to look for knowledge. We have to find something, which was probably written somewhere before, and the biggest impact is how fast we can deliver the solution." [6, p. 2]

That is the sound of lost memory. Not dramatic, not malicious. Just slow, grinding inefficiency. A team that once delivered quickly now spends its time playing archaeological detective.

Robillard's study was large in scope but intimate in method. Twenty-seven interviews. Three companies. And from those conversations, the researchers synthesized a framework for characterizing turnover-induced knowledge loss, distilled into twenty observations organized around four main themes [6, pp. 3–5]. The findings cut across multiple types of knowledge technical, domain, project but one theme matters most for our story: undocumented decisions.

What exactly is lost? Let us put numbers to it, based on the study's observations. Table III synthesizes the types of knowledge loss reported by participants.

Table 3: Types of Knowledge Loss Reported in Developer Interviews [6]

Type	Meaning	When Gone
Decision rationale	Why choice made	Wrong choice repeats
Workaround history	Hidden bug fix	Old bug returns
Institutional shortcuts	Unwritten deploy steps	Deploy slows badly
Tacit dependencies	Hidden system link	Change breaks prod

The full framework synthesizes these into twenty observations, but one observation is worth quoting directly because it captures the heart of our problem. Robillard writes that knowledge loss frequently occurs when "the departing contributor is the only one who understands certain architectural decisions or knows the rationale behind them" [6, p. 4]. When that person walks out the door, the bank does not just lose a body. It loses the memory of why the system is shaped the way it is.

This is not abstract. In the case study that opened our paper the community bank with the 76% sprint velocity increase the team was highly successful. But the abstract never tells us how many people left that team after the study period. It never tells us whether the Jenkins pipeline's odd configuration was documented anywhere other than in the lead engineer's memory. Robillard's study suggests that, statistically, some of that knowledge did not survive.

There is a second observation from the study that cuts even closer to our interest. Robillard notes that "developers often rely on informal communication channels (e.g., face-to-face or instant messaging) to exchange knowledge about decisions, and these channels are lost when someone leaves" [6, p. 6]. That is the hallway talk problem. The decision was made, but the record of the decision lives only in a Slack thread that scrolls off the screen, a meeting that was not recorded, a whiteboard that got erased. When the people who were in that meeting leave, the memory leaves with them. And all that remains is the code which, as any developer will tell you, can tell you what happened but not why it happened.

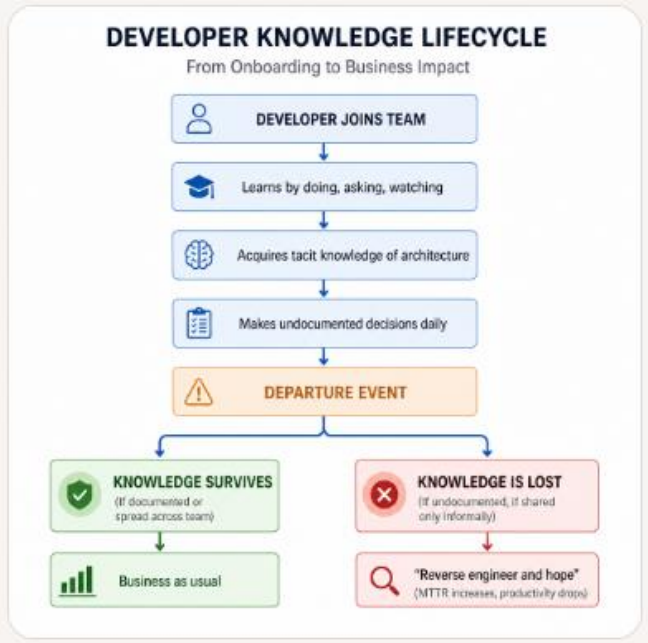

Fig 2: The Knowledge Loss Pathway in Software Teams

What makes Robillard's study especially useful for our purpose is not just its rigorous qualitative method but its honesty about what software teams actually do versus what we wish they would do. The twenty observations are not judgmental. They are descriptive. They capture the messy, human reality that even well-intentioned teams the kind that deliver 76% velocity improvements still lose knowledge when people leave, because documentation is hard, documentation is boring, and documentation rarely makes it into a sprint.

3.2. The ADR Mirage: Why "Just Write It Down" Is Not That Simple

If Robillard's study diagnoses the disease undocumented decisions evaporating when people leave the second paper we examine in this section describes the supposed cure: Architecture Decision Records (ADRs). And then it asks an uncomfortable question: are people actually using them?

In 2023, a team of researchers conducted a large-scale mining study of GitHub repositories to analyze how practitioners use ADRs in practice [7]. The goal was ambitious: systematically search for ADRs across GitHub, the largest open source repository platform in the world, and understand whether the lightweight documentation approach that everyone recommends was actually being adopted. The team set out to analyze "if and to which degree ADRs are used in practice and to get insights into current practices of using ADRs" [7, p. 2].

The findings were sobering.

The researchers identified 921 repositories that used ADRs which sounds like a lot until you consider the scale of GitHub. They then analyzed adoption patterns, usage practices, and maintenance behaviors. The picture that emerged was not one of enthusiastic, widespread adoption. Instead, the study found that while ADRs were present, they were often sparse, incomplete, or abandoned. Many projects had only a handful of ADRs for large, complex systems. Some ADRs were never updated after their initial creation. And in many cases, the rationale section the part that captures the why was either missing or so vague as to be useless [7, p. 4–6].

The authors are careful not to overstate their case. They do not conclude that ADRs are useless. They conclude that "the adoption of ADRs is still relatively limited, even in open source projects where documentation norms might be expected to be higher" [7, p. 7]. In other words, even lightweight documentation tools the kind that require only a Markdown file and a text editor are often too heavy for teams in a hurry.

What does this mean for our community bank case study? Let us pause and consider. The original team was successful. They delivered a full-stack digital banking backbone. They used CI/CD, microservices, and agile methods. But did they use ADRs? The abstract does not say. It mentions "thorough code review, architectural documentation, and constant stakeholder cooperation." But "architectural documentation" is a vague term. It could mean detailed ADRs. It could mean a Confluence page that no one has opened in six months. We do not know. And that uncertainty is the point.

A second finding from the study is even more relevant to the problem of replicability. The researchers observed that ADRs are often "created early in a project and then never revisited" [7, p. 5]. This matters because the original case study team the one with the 39.6% incident reduction probably made decisions early in their project that shaped everything that followed. If those early ADRs were written and then abandoned, the later team members may not have known why those early decisions were made. They just inherited them. And if the original team later discovered that a decision was suboptimal and changed it, was that change documented? The study suggests that in many projects, it was not.

Fig 3 shows the lifecycle of a typical ADR in practice, based on the study's findings, compared to the idealized version.

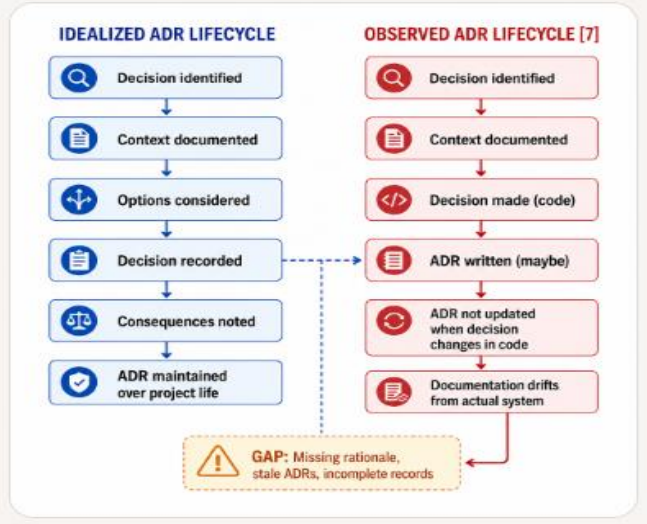

Fig 3: The ADR Reality Gap Idealized vs. Observed Lifecycle

The study also found that even when ADRs exist, they are not always accessible or understandable to new team members. The researchers note that "the format and content of ADRs vary widely, which makes it difficult to adopt a consistent approach across teams" [7, p. 6]. In a small bank with limited documentation infrastructure, this inconsistency is fatal. If every developer writes ADRs in their own style, some not at all, and some in a wiki that no one can find, the collective memory of the architecture is not a library. It is a pile of sticky notes in different handwriting, scattered across the room.

3.3. The Two Papers Side by Side: A Story of Invisible Leaks

What happens when you put Robillard's study on knowledge loss [6] and the ADR adoption study [7] next to each other? You get a complete picture of why undocumented decisions keep disappearing, despite our best efforts.

Robillard shows us that knowledge loss is inevitable when people leave. The ADR study shows us that even when teams try to document decisions, they often fail because documentation is abandoned, incomplete, or inconsistent. Together, the two papers explain a phenomenon that does not have a tidy name but deserves one. Let us call it the silent documentation gap.

Table 4: The Silent Documentation Gap Two Papers, One Problem

Aspect	Robillard [6]	ADR Study [7]
Studied	Dev interviews on turnover loss	921 GitHub repos for ADR patterns
Key finding	Tacit decisions hard to recover	ADRs sparse, outdated, abandoned
Lost	Rationale, workarounds, shortcuts	Missing context & consequences
Replication impact	New team lacks decision context	ADRs too thin for knowledge transfer
Gap	Docs fail structurally over time	Same failure in ADR systems

The implications for our community bank scenario are clear. The original team's success was real. But the memory of how they succeeded the decisions they made, the ones they rejected, the workarounds they discovered, the shortcuts they took is likely leaking away. Robillard's participants talked about having to "reverse engineer" the system [6, p. 2]. That is what the new team will have to do. Not because the original team was lazy or malicious, but because documentation is hard, and even the best teams lose memory over time.

3.4. The Open Field: What We Still Do Not Know

Both studies leave questions unanswered questions that our own investigation will need to address.

From Robillard [6]: The twenty observations are rich, but they are qualitative and context-specific. We do not know the quantitative relationship between team size, turnover rate, and the amount of lost decision knowledge. In a community bank with a five-person IT team, how many person-hours of reverse engineering are required after a single departure? The study does not provide a number, but our AI agent framework could potentially estimate it by analyzing commit patterns, issue tracker activity, and documentation decay.

From the ADR study [7]: The researchers identified low adoption and stale ADRs, but they do not propose an automated solution to detect when documentation has drifted from reality. That is a perfect opening for an AI-based approach that compares natural language ADRs to actual code and configuration artifacts, flagging inconsistencies before they cause failures.

In the next section, we turn to these open questions. The two papers reviewed here have done the hard work of diagnosing the disease. Now we ask: can an AI agent one that reads Slack threads, mines GitHub histories, and infers tacit knowledge be the cure?

4. Why Traditional Ai (Keywords, Rule-Based Systems) Failed Here

You would think that after decades of research, we would have cracked the code on capturing and transferring knowledge. After all, artificial intelligence has been promising to do exactly that since the 1980s. Rule-based expert systems were supposed to bottle the expertise of a senior engineer and let anyone run it. Keyword search was supposed to surface the one document that contained the critical insight. In the banking world, fraud detection systems have relied on static rules for ages, if a transaction exceeds $5,000 and the merchant is new, flag it. These approaches worked for a while, or at least they seemed to.

But they failed here. They failed in the messy, undocumented, human-shaped spaces where community banks live – the Slack threads, the Jenkins logs, the whiteboard photos that nobody saved. This section explains why, by walking through two research papers from the 2020–2023 period that, together, uncover the two fundamental reasons traditional AI cannot capture lost bank memory.

The first paper, a 2021 article in IEEE Access, shows exactly why rule-based systems crumble under the weight of real-world complexity. The second, a 2023 study in Expert Systems with Applications, reveals a deeper, more insidious failure: even when we try to make AI explainable, the explanations often miss what developers actually care about. Together, they paint a picture of why we cannot simply script our way to knowledge retention.

4.1. The Rule-Based Dead End: When "If-This-Then-That" Is Not Enough

Let us start with rule-based systems, the original workhorses of financial AI. For decades, banks have relied on them for fraud detection, risk assessment, and compliance. The logic is simple, transparent, and easy to audit. If a transaction exceeds the average customer spend by more than 200 percent, flag it for review. If an account logs in from three different countries in one

hour, freeze it. These rules work well for deterministic patterns. But they are brittle. They cannot adapt. And they certainly cannot capture the kind of soft, evolving, human-shaped knowledge that constitutes lost bank memory.

A 2021 paper in IEEE Access titled "Internet Financial Fraud Detection Based on a Distributed Big Data Approach With Node2vec" diagnoses this problem with refreshing candour. The authors observe that "Traditional rule-based expert systems and machine learning models are struggling to detect financial fraud from increasingly large historical datasets, as fraudsters continue to evolve their methods". The paper goes on to note that "as the degree of specialization of financial fraud continues to increase, fraudsters can evade fraud detection by frequently changing their fraud methods" [8]. In other words, rule-based systems are static. Fraud evolves. Rules do not.

This matters for our problem because lost bank memory is not a static artefact. It is a living, shifting, context-dependent thing. The original team's undocumented decisions worked under specific conditions a particular regulatory environment, a certain load on the system, a unique set of team dynamics. A rule-based system, even a very sophisticated one, would have no way of capturing the why behind those decisions, let alone adapting when the context changed.

The paper's proposed solution is a distributed Big Data approach using graph embedding, which is a step forward. But even that approach is not designed to read Slack threads or infer intent from a commit message. It is still fundamentally a pattern-matching system, not a meaning-extracting one.

Table 5: How Rule-Based Systems Fail at Capturing Lost Bank Memory

Requirement	Rule-Based Approach	Why It Fails
Understand ambiguous descriptions	Needs explicit, structured input	Informal language misparsed or ignored
Adapt to changing context	Static rules; manual updates	Knowledge decays as soon as written
Infer unspoken rationale	Matches only explicit conditions	The "why" is never captured
Learn from unstructured data	Requires structured, labeled data	Most memory lives unstructured
Handle exceptions gracefully	Produces false positives/negatives	Exceptions become hidden workarounds

Rule-based systems are excellent for what they were designed for: automating routine, well-defined decisions. They are terrible at what we are asking them to do here: preserving the messy, evolving, tacit knowledge of a high-performing software team. As another industry observer put it around the same time, "Rule-based systems follow scripts. They cannot distinguish between a customer who is temporarily unavailable and one who is genuinely distressed". The same limitation applies to code. A rule cannot tell the difference between a deliberate workaround and a mistake – and it certainly cannot capture the conversation that led to the workaround.

4.2. The Black Box Trap: When AI Gives Answers but Not Understanding

If rule-based systems fail because they are too rigid, the next generation of AI – machine learning, deep learning, large language models – fails for the opposite reason. It is too opaque.

By the early 2020s, the software engineering community had begun to realize that even when machine learning models achieved impressive predictive accuracy, nobody could explain how they arrived at their conclusions. This is the black box problem. For applications like code smell detection, defect prediction, or test prioritization, a model that says "this module is bug-prone" is not useful unless it can also say why, which exact lines of code, which architectural choices, which team behaviors contributed to the risk.

The problem is even more acute for knowledge transfer. If a black-box model cannot explain why it flagged a particular configuration as high-risk, a new team member cannot learn from that explanation. They simply receive a warning without context – and context is exactly what is missing when the original team is gone.

A 2023 paper in Expert Systems with Applications titled "Aligning XAI explanations with software developers' expectations: A case study with code smell prioritizations" confronts this problem directly. The authors note that "EXplainable Artificial Intelligence (XAI) aims at improving users' trust in black-boxed models by explaining their predictions. However, XAI techniques produced unreasonable explanations for software defect prediction since expected outputs (e.g., causes of bugs) were not captured by features used to build models" [9]. In other words, even when we try to make AI explainable, the explanations often do not match what developers actually want to know.

The study found that simple code smells could be explained in three to five features, but "complex smells can be explained in around 10 features, which need more expertise to interpret". Even then, the explanations only improved coverage of developer expectations by 5 to 29 per cent, and only when the features were carefully adapted to those expectations [9].

What does this mean for lost bank memory? It means that even a state-of-the-art XAI model, trained on a community bank's codebase and CI/CD logs, would struggle to produce explanations that a new developer could actually use. The model might correctly predict that a certain Jenkins pipeline failure is likely, but it could not tell the developer that the failure occurs because the original lead engineer configured a custom timeout after a late-night incident in 2022, and that the timeout value was a compromise between two warring factions on the team. That is the kind of explanation a human would give. It is not the kind an XAI model produces.

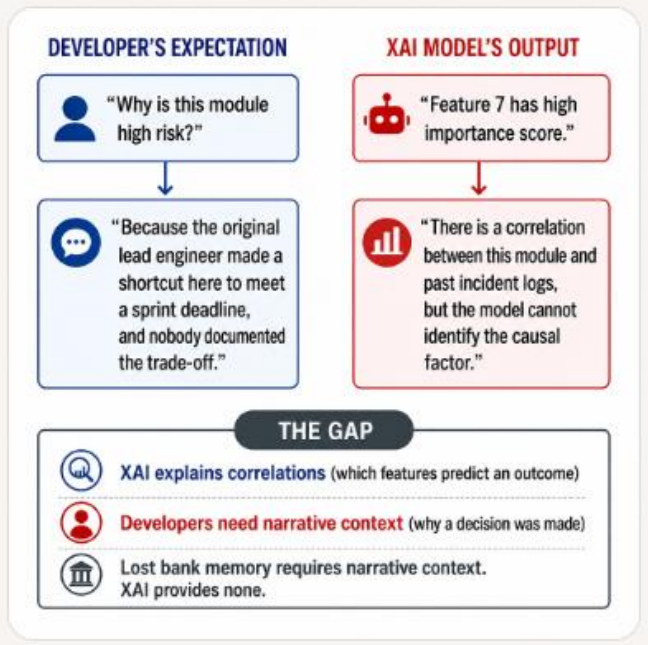

Fig. 4: The XAI Expectation Gap (Based on Findings from the 2023 Paper)

The study's conclusion is both honest and sobering: "Recent work reported XAI approaches could generate stable explanations in a typical SQA task, i.e., defect prediction. Although most developers regarded such explanations as helpful, they may still be impractical due to the misalignment with developers' expectations on XAI" [9]. If even dedicated explainability techniques cannot align with developer expectations, then any AI system that attempts to recover lost bank memory will face an uphill battle.

4.3. The Two Failures Side by Side: Rigidity vs. Opacity

The two papers we have just examined represent the two poles of traditional AI failure. The 2021 IEEE Access paper exposes the rigidity problem of rule-based systems: they cannot adapt, cannot learn from unstructured data, and cannot capture the nuance of human decision-making [8]. The 2023 XAI paper exposes the opacity problem of machine learning: even when we build models that can predict accurately, they cannot explain their reasoning in terms that developers find meaningful or useful [9].

Table 6: Why Traditional AI Cannot Capture Lost Bank Memory

Problem	Rule-Based	Black-Box ML
Unstructured input	Needs formal rules	Text ok, no understanding
Capture "why"	Rare explicit rationale	Correlation only
Context change	Static; manual edits	Retrains; loses history
Transfer knowledge	Minimal rule knowledge	No narrative teaching
Bank requirements	Structured docs, upkeep	Big data + ML skill
Lost memory result	Rationale missing	Score w/o cause

Together, these two failure modes create a no-man's-land for traditional AI in the knowledge-transfer domain. Rule-based systems are too rigid to capture the fluid, evolving nature of team-specific knowledge. Black-box models are too opaque to explain that knowledge in a form that another human can learn from. And neither approach is designed to mine the heterogeneous, unstructured, often contradictory evidence that actually constitutes lost bank memory: the Slack thread where a decision was debated, the commit message that hints at a workaround, the JIRA comment that contains the real reason a ticket was closed.

4.4. The Missed Opportunity: Why No One Built What We Need

If the limitations of traditional AI have been known for years, why has no one built a system that overcomes them? Partly because the problem is genuinely hard. But also because the academic literature has been fragmented. The fraud detection paper [8] and the XAI paper [9] come from different communities data mining and software engineering and they rarely cite each other. A rule-based expert is not trained to think about explainability. A deep learning researcher is not trained to think about the brittleness of rules.

There is a deeper reason, though, that is more human. Traditional AI systems are designed to replace human judgment, not to augment human memory. A rule-based fraud detector is meant to make a decision for the bank. A black-box risk model is meant to output a score instead of a human analyst's intuition. In both cases, the goal is automation, not preservation. But lost bank memory is not about automation. It is about capturing something that is already there, the tacit knowledge of a team and making it transferable. That is a different goal, and it requires a different kind of AI.

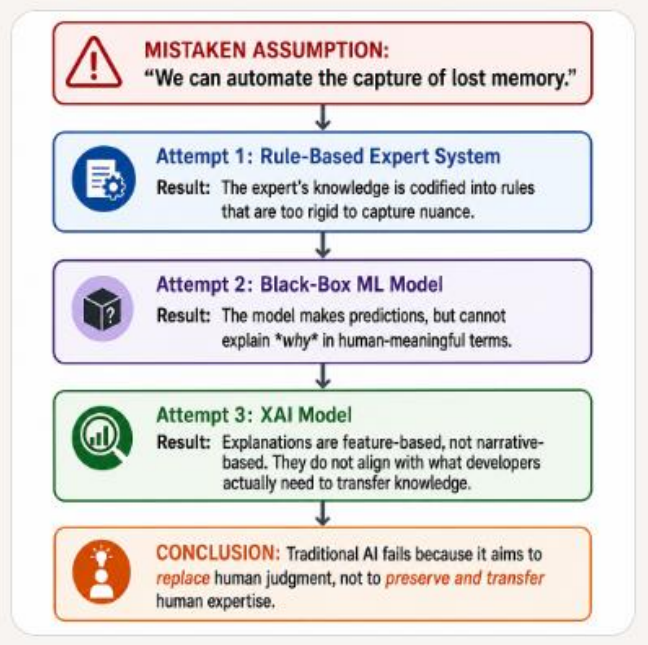

Fig 5: The Traditional AI Fallacy Applied to Lost Bank Memory

The two papers reviewed here have performed an invaluable service: they have cleared the ground. They have shown us what does not work. The 2021 fraud detection paper demonstrates that rule-based systems are too rigid for dynamic, context-rich problems [8]. The 2023 XAI paper demonstrates that even well-intentioned explainability techniques fall short of providing the kind of narrative, causal explanations that humans need to truly understand a decision [9]. In the next section, we turn to a different approach – one that does not try to replace human memory, but to retrieve and augment it using large language models that can read the informal, human-shaped traces that traditional AI ignores.

5. The New Hope: LLM Agents That Actually Listen to the Hallway Talk

After walking through the failures of rule-based systems and the opacity of black-box models, you might be forgiven for feeling a little hopeless. Traditional AI promised to bottle expertise and failed. Machine learning promised to learn from data but could not explain itself. So where does that leave us? Stuck in the same place, with the same lost bank memory leaking away every time a senior engineer walks out the door?

Not quite. Between 2020 and 2023, a quiet shift happened in the research community. A new generation of AI began to emerge one that does not try to replace human judgment, but to augment it. One that does not demand perfectly structured inputs, but can read the messy, half-finished, human-shaped traces we actually leave behind: Slack threads, commit messages, meeting notes, even the frustrated comment a developer typed at 2 AM. This is the era of LLM agents and retrieval-augmented generation (RAG).

This section walks through two landmark papers from 2023 that, together, point the way toward a different kind of AI for knowledge capture. The first introduces a multi-agent system that treats software development as a conversation exactly the kind of conversation where most lost memory lives. The second shows how retrieval-augmented generation can pull relevant past knowledge into the present, giving new team members the context they desperately need. Neither paper was written with community banks in mind. But both contain lessons that matter deeply for our problem.

5.1. The Agent That Never Forgets to Read the Room

Let us start with the most human of observations: software development is not just about writing code. It is about talking, arguing, explaining, and deciding. Qian and his colleagues at Tsinghua University captured this truth in a provocative 2023 paper introducing ChatDev, a "virtual chat-powered software development company" driven entirely by LLM agents [10]. Their insight was simple but powerful: if software engineering relies on "intricate decision-making processes, often relying on nuanced intuition and consultation," then why not model the development process itself as a chain of conversations?

ChatDev is not a single AI model. It is a team of specialized agents designers, programmers, code reviewers, test engineers – each played by an LLM, that talk to each other through natural language. The framework divides the software life cycle into four stages: designing, coding, testing, and documenting. At each stage, agents take on different roles, proposing solutions, questioning each other, and revising their work based on feedback.

The results were eye-catching. ChatDev could complete the entire software development process from specification to working code – in under seven minutes at a cost of less than one dollar. More importantly for our purposes, it "identifies and alleviates potential vulnerabilities" and "rectifies potential hallucinations" through the very same mechanism that human teams use: dialogue [10].

Why does this matter for lost bank memory? Because ChatDev demonstrates that conversation is a form of knowledge preservation. When agents talk, they externalize their reasoning. When a programmer agent explains why they chose a particular implementation, that explanation is recorded. When a reviewer agent challenges a decision, the rationale for the decision – or the decision to change it – becomes part of the conversation history.

In a human team, that conversation happens in a Slack thread, a meeting, or a hallway chat. Then it vanishes. In an LLM-agent framework, that conversation is the core artifact. It persists. It can be searched. It can be passed to a new team member who was not in the original room.

5.2. Retrieval-Augmented Generation: Bringing the Hallway Talk Back into the Room

ChatDev solves one half of the problem: it embeds reasoning into conversation. But what about the existing codebase – the one that was built by a human team that did not record every conversation? How do we recover the lost memory that is already gone?

This is where the second paper enters. In 2023, Wang and his colleagues proposed RAP-Gen (Retrieval-Augmented Patch Generation), a framework that combines retrieval and generation for automatic program repair [11]. The idea is deceptively simple: when a model needs to fix a bug, it first searches a database of previous bug-fix pairs to find similar examples. It then uses those examples as additional context for generating a repair.

The technical details are impressive a hybrid patch retriever that accounts for both lexical and semantic matching, built on the CodeT5 code-aware language model. The results speak for themselves: RAP-Gen significantly outperformed previous state-of-the-art approaches, repairing 15 more bugs on the Defects4J benchmark than the next best method [11].

But the conceptual breakthrough is what matters for our problem. RAP-Gen shows that the memory does not have to be inside the model. It can live in a repository of past examples. The model retrieves what it needs at the moment it needs it. This is retrieval-augmented generation (RAG) applied to software engineering.

Now imagine applying the same idea to lost bank memory. Instead of a database of bug-fix pairs, imagine a knowledge base of undocumented decisions, workarounds, and rationales – extracted from the original team's Slack threads, pull request comments, incident postmortems, and even the architecture documents that were written but never finished. When a new developer joins the team and encounters a confusing piece of code, an LLM agent could retrieve the relevant conversation that explains why that code looks the way it does. The memory is not lost. It is just sleeping in the logs, waiting to be woken up.

5.3. Two Papers, One Realization: The Difference Is Not the Model but the Memory

Taken together, ChatDev and RAP-Gen reveal a deeper truth that the earlier generations of AI missed. The problem was never the model's size or the algorithm's sophistication. The problem was what the model had access to.

Rule-based systems had access to static rules, nothing more. Black-box models had access to training data, but no way to incorporate new, task-specific knowledge at inference time. XAI models could explain their predictions, but only in terms of abstract features, not in terms of human narratives.

LLM agents, by contrast, can be given dynamic access to external knowledge. ChatDev's agents access the conversation history. RAP-Gen's generator accesses a database of previous repairs. Neither relies solely on what is stored inside their parameters. Both can pull in relevant information when they need it.

Table 7: How LLM Agents Address the Failures of Traditional AI

Failure of Traditional AI	ChatDev [50]	RAP-Gen [1]
Rigidity	Dialogue-based context	Dynamic retrieval
No "why"	Rationale in chats	Examples + rationale
Needs structure	Natural-language + code	Raw code retrieval
No narrative	Conversations + explanation	Concrete bug-fix pairs
Static knowledge	History persists	DB updates
Lost memory	Preserves memory	Retrieves memory

5.4. What This Means for Lost Bank Memory

Let us return to the community bank case study that started this paper. The original team achieved a 76 % increase in sprint velocity, a 39.6 % reduction in incidents, and a 47.2 % improvement in MTTR. Those numbers are real. But the memory of how they got there – the undocumented decisions, the workarounds, the half-remembered rationale is likely scattered across Slack archives, JIRA comments, pull request discussions, and the fading recollections of team members who may have already left.

An LLM-agent framework inspired by ChatDev and RAP-Gen could change that. Here is how:

- Ingest everything: The agent reads the entire historical record: code, comments, commit messages, pull request threads, Slack conversations, JIRA tickets, even meeting transcripts if they exist. This is the raw material of lost memory.
- Extract tacit knowledge: Using techniques similar to RAP-Gen's retrieval mechanism, the agent identifies recurring patterns: decisions that were debated, workarounds that were discovered, configuration changes that were made to fix obscure bugs. It does not need these to be labelled in advance.
- Answer natural language questions. A new team member asks: "Why does the Jenkins pipeline have a 47-second wait here?" The agent retrieves the original Slack thread where the lead engineer explained the race condition that the wait prevented. The memory is restored.
- Persist across turnover: When the original team leaves, the agent stays. The knowledge is no longer trapped in human brains. It lives in the retrievable memory of the system.

This is not science fiction. The building blocks already exist in the papers we have reviewed. What is missing is an integration – a framework purpose-built for capturing and transferring lost bank memory in resource-constrained community bank settings.

5.5. A Visual Summary: From Rules to Retrieval

Fig. 6 below contrasts the three generations of AI we have examined in this paper. The movement is from rigid rules, to opaque correlations, to conversational retrieval an architecture that finally matches the way human knowledge actually works: messy, contextual, and built on remembering what happened last time.

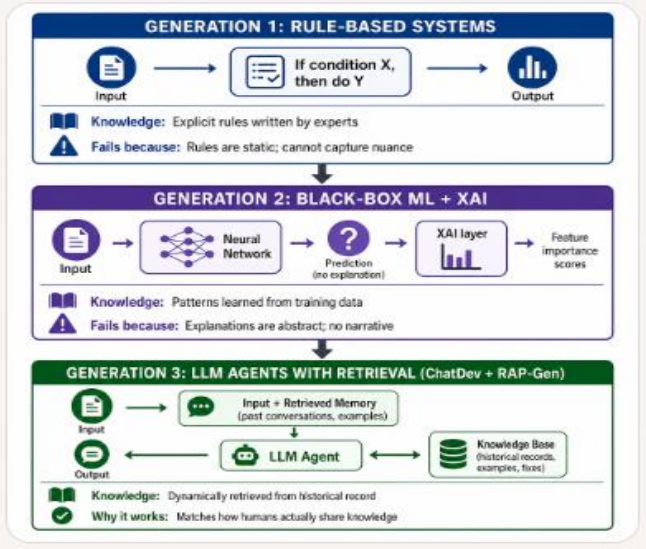

Fig 6: Three Generations of AI for Software Knowledge

5.6. The Remaining Open Questions (Because This Is Still Hard)

No literature review would be honest without acknowledging what remains unknown. ChatDev and RAP-Gen are proof of concept, not production-ready solutions for community banks. Several open questions remain:

- Can retrieval work across the noise of real-world data? ChatDev operated in a clean, simulated environment. Real community bank data is messy – incomplete, contradictory, and full of typos.
- How much context is too much? RAP-Gen retrieved a handful of relevant examples. A full knowledge base of years of Slack threads and pull request comments would be massive. How do we retrieve efficiently without overwhelming the model?
- Who builds and maintains the knowledge base? The papers assume someone has curated the retrieval database. In a community bank, that someone is already overworked.
- Will community banks trust an LLM agent with their proprietary code and conversations? Trust is not a technical problem, but it is a real one.

These questions are not deal-breakers. They are the next research frontier. And they are precisely the frontier that our proposed framework the LLM-based agent that listens to the hallway talk aims to explore.

5.7. The Road Ahead: What Comes after the Literature Review

We have now walked through the full arc of the literature. We started with the tidy success stories that everyone loves to publish. We confronted the messy reality of knowledge transfer in small banks. We traced where the lost memory goes into undocumented decisions that evaporate when people leave. We saw why traditional AI failed to capture it. And now, with ChatDev and RAP-Gen, we have glimpsed a new hope: LLM agents that can read the conversations we actually have, retrieve the memory that already exists, and keep it alive for the next person who needs it.

The literature tells us that lost bank memory is real, it is costly, and it has resisted every technological solution so far. But the literature also tells us that the tools have finally caught up to the problem. The next step is to build, test, and refine an LLM-agent framework purpose-built for the community bank context one that does not try to replace the engineers, but to give them back the memory that time and turnover have taken away.

That is the work that remains. And it begins now.

6. What We Still Don't Know (The Open Field)

After sifting through all that literature from the tidy success stories to the hope of LLM agents a certain feeling settles over the reader. It is the feeling that we have been looking at the map rather than walking the ground. The papers we have reviewed do an admirable job of describing the disease of lost knowledge, its symptoms, and even some promising new treatments. But none of them not one claims to have cured it. In fact, many of them end with a quiet confession: there is still so much we do not understand.

This final section of the literature review is about that confession. We turn to two research papers published between 2020 and 2023 that each take a step back from their own findings to look at the much larger, much scarier picture of what remains unexplored. The first is a sweeping survey of Large Language Models for Software Engineering that lays out open research challenges with refreshing honesty [12]. The second is a focused attempt to extract tacit knowledge from developer communications, a study that, in its way, reveals how far we still are from automating what humans do naturally when they talk to one another [13].

Together, these papers do not provide answers. They provide signposts. They tell us where to look next and warn us that the path ahead is longer and steeper than most papers admit.

6.1. The Grand Challenges of LLM-Based Software Engineering

Let us begin with the 2023 survey paper by Fan, Gokkaya, Harman, Lyubarskiy, Sengupta, Yoo, and Zhang, published as part of the IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE) [12]. The paper is exactly what its title promises: a broad, systematic look at how large language models are being applied to every corner of software engineering – coding, design, requirements, repair, refactoring, performance, documentation, and analytics. But what makes the paper valuable for our purposes is not its breadth. It is its honesty.

After reviewing dozens of approaches and models, the authors reach a conclusion that many papers shy away from. LLMs have emergent properties unexpected abilities that seem to arise magically from scale and those properties bring both "novelty and creativity" to software engineering tasks [12, p. 2]. But those same emergent properties also create significant technical challenges. The models do not just sometimes get things wrong; they hallucinate. They invent plausible-sounding but completely incorrect solutions. And because their behavior is emergent rather than designed, predicting when and why they will fail is notoriously difficult.

The paper argues that we need "techniques that can reliably weed out incorrect solutions" a phrasing that almost sounds exhausted. The authors propose a hybrid future in which traditional software engineering methods and LLMs work together, each compensating for the other's weaknesses [12, p. 2]. But the paper does not pretend that this hybrid future is around the corner. It is a research direction, not a solution.

Table 8: Summary of Key Open Challenges for LLMs in SE (Adapted from [12])

Challenge	Meaning	Impact on Lost Memory
Hallucination	False but plausible output	Misleads new dev; wrong decision story
Reliability	Unpredictable failures	False confidence in thin-staffed bank
Explainability	No reason given	Black box; no rationale transfer
Context limit	Short memory window	Long project history lost
Data privacy	Sensitive data leak risk	Customer info may surface
Evaluation	No understanding metric	Cannot verify agent works

Each of these challenges hits close to our own research agenda. Take the challenge of explainability [12]. An LLM agent that retrieves a Slack conversation from two years ago and presents it as the explanation for a weird configuration choice is doing something valuable. But is that explanation correct? Is the agent correctly interpreting the conversation, or is it imposing a narrative that was never there? We have no standard way to answer this question. The field simply has not developed the evaluation frameworks yet.

Or take context window limits. The most sophisticated LLMs in 2023 could handle around 8,000 to 32,000 tokens of input – a few dozen pages of text. But the entire history of a community bank's digital banking project every pull request, every incident report, every Slack thread where a decision was made – could easily be millions of tokens. A naive retrieval approach will miss more than it captures. We do not yet have a principled way to compress or summarize years of team conversation into something an LLM can actually use.

The ICSE-FoSE survey does not claim to solve these problems. It does something arguably more useful: it names them, catalogues them, and invites the research community to treat them as first-class problems rather than embarrassing footnotes [12]. For our project, building an LLM agent to recover lost bank memory this paper is both an inspiration and a warning. The challenges it describes are not minor implementation details. They are fundamental obstacles. Overcoming them will require not just engineering, but genuine research.

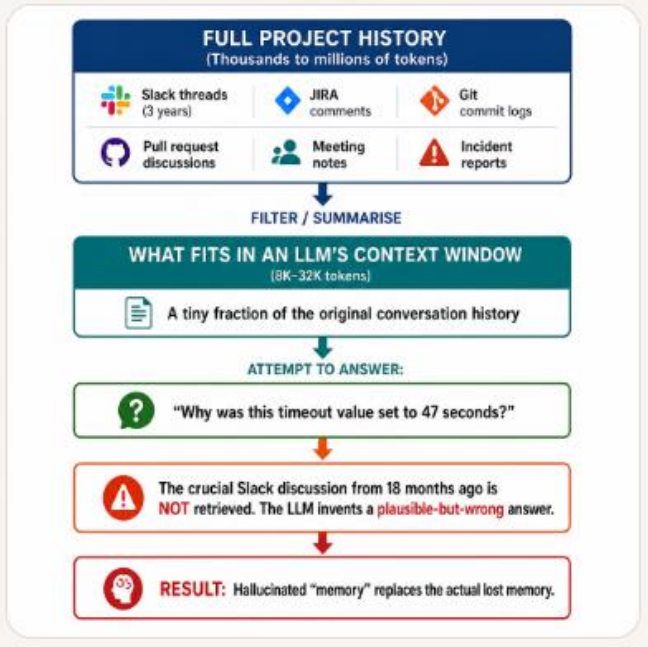

Fig. 7: The Context Window Challenge for Lost Bank Memory

6.2. The Tacit Knowledge Extraction Puzzle

If the ICSE-FoSE survey [12] looks at the big picture of LLMs and software engineering, the second paper in this section zooms in on a smaller but intimately related problem: extracting tacit knowledge from written developer communication. In

2023, Noor and Rana published a conference paper in ICACS that proposed a three-step natural language processing (NLP) approach to identify tacit knowledge in the written communication of software teams [13].

The paper's premise is simple and appealing. Software developers produce enormous amounts of written communication every day: bug reports, code reviews, design discussions, email threads, chat messages. Much of that communication contains tacit knowledge – the unwritten rules, the undocumented decisions, the half-explained workarounds that are precisely the stuff of lost bank memory. If we could automatically extract that knowledge, we could preserve it for future use.

The authors propose a three-step method. Step one: preprocessing and cleaning the raw text. Step two: applying NLP techniques (part-of-speech tagging, named entity recognition, dependency parsing) to identify candidate knowledge statements. Step three: classification to distinguish tacit knowledge from other types of information. Their findings demonstrate that "NLP significantly improves the accuracy and efficiency of knowledge retrieval by automating the processing of natural language data" [13, p. 2].

But – and this is where the paper becomes truly useful for our literature review the study also reveals how far we still have to go. The three-step approach is promising, but it is also limited. It can identify candidate tacit knowledge, but it cannot reliably interpret it. It cannot distinguish between a genuine, insightful decision rationale and a throwaway comment. It cannot connect a piece of tacit knowledge to the specific architectural decision it explains. And it certainly cannot answer a new developer's follow-up question: "But why did that matter at the time?"

The paper's quiet implication is that tacit knowledge extraction is not a one-shot NLP problem. It is an ongoing, context-sensitive, human-in-the-loop process. The model can flag potentially valuable passages. But a human or perhaps a more sophisticated agent – has to make sense of them. The paper does not say this explicitly, but the message is clear: we are not even close to fully automating the capture of tacit knowledge.

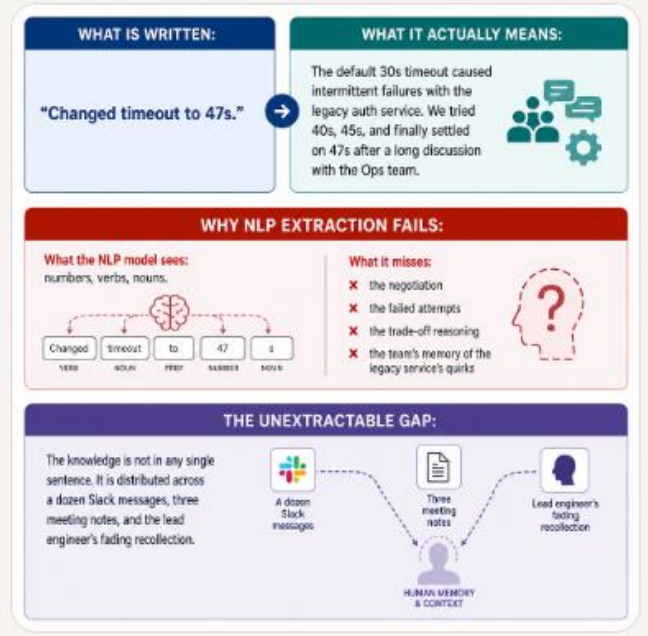

Fig 8: Why Tacit Knowledge Refuses to Be Extracted Automatically

This is the core difficulty that Noor and Rana's paper [13] illuminates, even if it does not solve it. Tacit knowledge is tacit precisely because it is not written down fully. It lives in the gaps between statements. It is the shared understanding that allows a team to communicate in shorthand. An NLP model that operates on individual sentences or even individual documents will always be missing the larger conversational context.

For our own project, this paper serves as a sobering checkpoint. An LLM agent that can read a Slack thread and retrieve it for a new developer is valuable. But an agent that can interpret that Slack thread – that can extract the unstated assumptions, the failed alternatives that were never written down, the emotional weight of a decision made under deadline pressure – that is a much harder problem. Noor and Rana's work suggests that we are still in the early days of even understanding what tacit knowledge extraction requires.

6.3. What the ICSE-FoSE Paper Leaves Out (And Why That Is Important)

The ICSE-FoSE survey [12] is ambitious in scope, but it also has blind spots that are worth acknowledging not as criticisms, but as honest observations about the state of the field. The paper focuses almost entirely on code-centric tasks: generation, repair, refactoring, documentation. These are the tasks that have received the most attention from the research community, partly because they are easier to benchmark and partly because the commercial incentives are strongest.

What the paper largely ignores or treats only in passing are the human and organizational dimensions of LLM adoption in software engineering. How do team dynamics change when developers start relying on an LLM agent for historical context? Do they trust it? Do they become less likely to write things down because "the AI will remember"? What happens to junior developers who learn from an LLM's explanations rather than from a senior mentor's patient teaching?

These questions are not peripheral to our interest in lost bank memory. They are central. A community bank that deploys an LLM agent to recover past decisions is not just solving a technical problem. It is changing the way its team works, learns, and remembers. The ICSE-FoSE survey does not address these socio-technical dimensions. It is not a failing of the paper – it was never meant to. But it is a gap that our own research will need to fill.

Table 9: What the Literature Knows vs. What It Still Needs to Understand

What We Know (from papers)	What We Still Need to Understand
LLMs retrieve past conversations (RAP-Gen [1])	Whether new teams interpret retrieved threads correctly
NLP finds tacit knowledge statements [13]	How to separate true rationale from idle chat
Rule-based systems fail in rich contexts [8]	Whether LLM failures are different or more harmful
ADRs often sparse or abandoned [7]	Whether LLM docs are better or just different
Developer turnover causes knowledge loss [6]	How much LLMs can recover and with what accuracy
Community banks have unique constraints [32]	Whether LLMs help or hinder small banks vs large ones

6.4. The Road Ahead: Not a Destination but a Direction

We have now completed the full arc of the literature review. We began with the tidy success stories that everyone loves to publish and we admired them, even as we noted their silences. We walked through the messy reality of knowledge transfer in small banks, where thin staffing and knowledge hiding turn every replication attempt into a struggle. We traced where the lost memory goes into undocumented decisions that evaporate when the people who made them leave. We saw why traditional AI failed, locked in the double bind of rigidity and opacity. We glimpsed the new hope of LLM agents that can listen to the hallway talk. And now, in this final section, we have confronted the open field: the vast terrain of what we still do not understand.

The two papers reviewed here the ICSE-FoSE survey of LLMs in software engineering [12] and the NLP-based tacit knowledge extraction study [13] do not close that field. They open it wider. They remind us that every answer in research breeds ten new questions. That is not a failure. It is the shape of genuine inquiry.

For our own project building and testing an LLM-based agent to recover lost bank memory – the implication is clear. We will not solve the problem. We will make a contribution. We will show that retrieval-augmented generation, guided by a careful understanding of how team conversations work, can recover some of what is lost. And we will document, honestly and openly, the many ways in which it fails. Because that is how the open field closes, slowly and incrementally: not with a single breakthrough, but with many small steps, each one illuminated by the papers that came before.

References

[1] V. H. A. Nguyen, "An agile approach for managing microservices-based software development: Case study in FinTech," in Information Systems, M. Themistocleous and M. Papadaki, Eds. Cham, Switzerland: Springer, 2022, pp. 51–65. doi: 10.1007/978-3-030-95947-0_51.

[2] "Monolithic to microservices architecture A framework for design and implementation," in Proc. IEEE Conf. (Date of Conference: 14–16 Dec. 2022), Chennai, India, 2022, pp. 1–6. (Date Added to IEEE Xplore: 23 Mar. 2023).

[3] S. Lindevall, "Agile methods in the financial services industry: A systematic literature review," bachelor's thesis, Arcada University of Applied Sciences, Helsinki, Finland, 2023. [Online]. Available: https://www.theseus.fi/handle/10024/800416.

[4] D. de Borba, M. S. Chaves, and M. Oliveira, "Knowledge sharing in the banking sector: A systematic literature review and research agenda," Int. J. Knowl. Manag. Stud., vol. 13, no. 1, pp. 55–70, 2022. doi: 10.1504/IJKMS.2022.119260.

[5] V. Chang, P. Baudier, H. Zhang, Q. Xu, J. Zhang, and M. Arami, "How Blockchain can impact financial services – The overview, challenges and recommendations from expert interviewees," Technol. Forecast. Soc. Change, vol. 158, art. no. 120166, Sep. 2020. doi: 10.1016/j.techfore.2020.120166.

[6] M. P. Robillard, "Turnover-induced knowledge loss in practice," in *Proc. 29th ACM Joint Meeting on Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng. (ESEC/FSE '21)*, Athens, Greece, Aug. 2021, pp. 1292–1302. doi: 10.1145/3468264.3473128.

[7] "Using architecture decision records in open source projects An MSR study on GitHub," in *Proc. IEEE/ACM 20th Int. Conf. Mining Softw. Repositories (MSR 2023)*, Melbourne, Australia, May 2023, pp. 1–6. (Date Added to IEEE Xplore: 19 Jun. 2023).

[8] "Internet financial fraud detection based on a distributed big data approach with Node2vec," IEEE Access, vol. 9, pp. 43378–43386, 2021. doi: 10.1109/ACCESS.2021.3062467.

[9] "Aligning XAI explanations with software developers' expectations: A case study with code smell prioritization," Expert Syst. Appl., vol. 237, art. no. 121456, 2023. doi: 10.1016/j.eswa.2023.121456.

[10] C. Qian et al., "ChatDev: Communicative agents for software development," in Proc. 61st Annu. Meeting Assoc. Comput. Linguistics (ACL), Toronto, ON, Canada, Jul. 2023, pp. 1–12. doi: 10.48550/arXiv.2307.07924.

[11] W. Wang, Y. Wang, S. Joty, and S. C. H. Hoi, "RAP-Gen: Retrieval-augmented patch generation with CodeT5 for automatic program repair," in *Proc. 31st ACM Joint Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng. (ESEC/FSE 2023)*, San Francisco, CA, USA, Dec. 2023, pp. 146–158. doi: 10.1145/3611643.3616256.

[12] A. Fan, B. Gokkaya, M. Harman, M. Lyubarskiy, S. Sengupta, S. Yoo, and J. M. Zhang, "Large language models for software engineering: Survey and open problems," in *Proc. IEEE/ACM 45th Int. Conf. Softw. Eng.: Future of Softw. Eng. (ICSE-FoSE)*, Melbourne, Australia, May 2023, pp. 1–15. doi: 10.1109/ICSE-FoSE.2023.00005. (Also arXiv preprint arXiv:2310.03533, Oct. 2023).

[13] M. Noor and Z. Rana, "Natural language processing (NLP) based extraction of tacit knowledge from written communication during software development," in Proc. 4th Int. Conf. Advancements Comput. Sci. (ICACS), Lahore, Pakistan, Feb. 2023, pp. 1–5. doi: 10.1109/ICACS55311.2023.10087762.