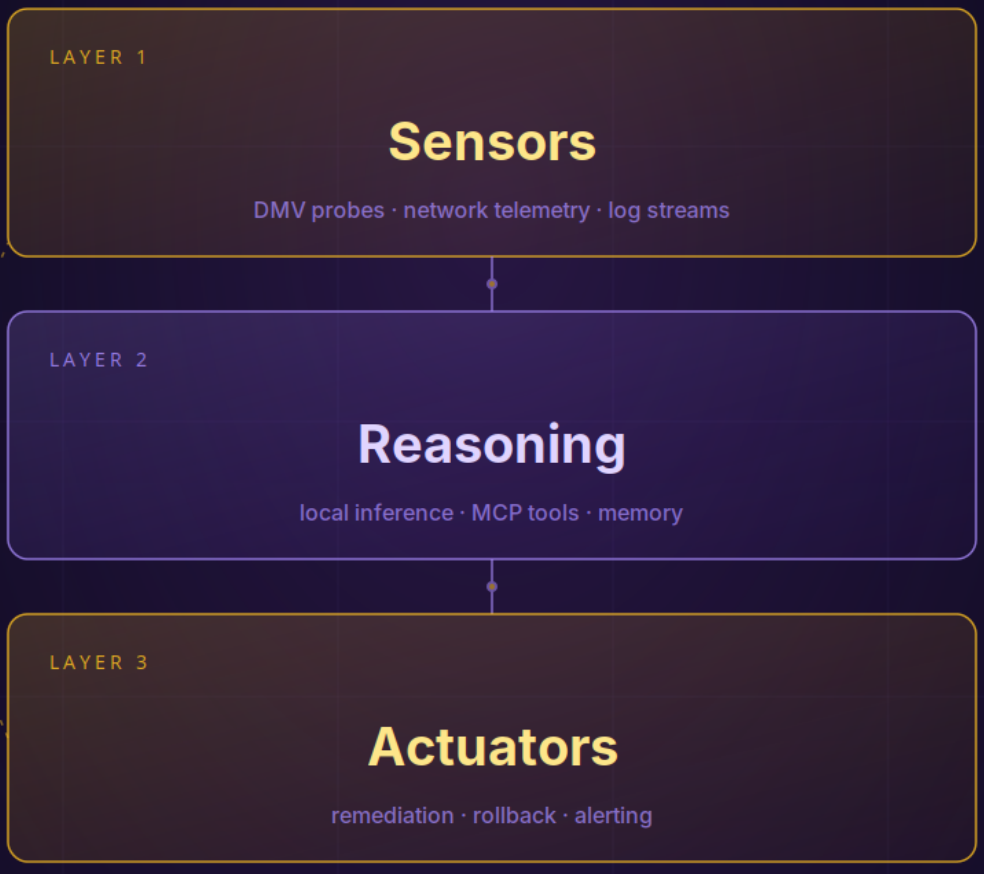
The Birth of Bob

An Autonomous Agent for SQL Server Self-Healing

Ward T. Minson

The Birth of Bob

An Autonomous Agent for SQL Server Self-Healing



Contents

The Birth of Bob

An Autonomous Agent for SQL Server Self-Healing and Network Monitoring

Ward T. Minson

For every DBA who has opened a 200-page health report at midnight and wondered if there was a better way. There is.

How to Read This Book

This book has three parts. Part One is the origin story: where Bob came from, and why a DBA with nearly twenty years behind him would spend eighteen months building a local AI agent when cloud tools were already sitting on his workstation. Part Two is the build: infrastructure, models, the agent architecture, the protocol that ties it together. Part Three is the road ahead: self-healing, community, and what it means to scale sovereignty beyond one server room.

If you are a senior DBA who wants to know whether any of this is safe to run near regulated data, start at Chapter 1 and read straight through. If you are a developer who wants to wire a new capability into Bob this weekend, jump to Chapter 7 and come back to the rest later. If you run Proxmox at home and your question is hardware, Chapter 4 has the sizing table you want.

The code samples in Part Two are tied to SHAs in the live Bob repository at `hyp3rsoft/bob` on the internal Gitea. The placeholder `<sha>` appears where the M15 code-sample pass will fill in verified hashes against the production system. Every infrastructure fact, IP address, hostname, and VM ID matches production as of the date on this manuscript. If something has drifted, `make audit` will tell you before the book ships.

Agent Vocabulary

A single-page glossary for the vibe coder who needs to talk to a DBA, and the DBA who needs to talk to an agent. Five terms. No hand-waving.

Query plan. The execution strategy SQL Server chooses when it runs a query. Represented as an XML tree or a graphical diagram in SSMS, it shows which indexes the optimizer selected, how it will join tables, where row-count estimates went wrong, and which operators will consume the most CPU or I/O. A DBA reads query plans to find why a query is slow. Bob reads them to decide whether a proposed fix is safe to apply. A plan with an estimated row count of 1 against a table that actually has 8 million rows is the shape of a bad plan. You learn to recognize the shapes.

Wait stats. When a SQL Server thread cannot run, it waits. The `sys.dm_os_wait_stats` DMV records what it waited for and how long. A session waiting on `PAGEIOLATCH_SH` has a disk I/O problem. A session waiting on `LCK_M_X` has a locking problem. `CXPACKET` waits at scale suggest a parallelism configuration issue. Bob's sensor layer polls wait stats continuously. They are the primary signal that tells the agent something has changed.

Deadlock graph. Two transactions, each holding a lock the other needs. Neither can proceed. SQL Server kills one and logs the event as an XML deadlock graph in the system health session. A single deadlock is a curiosity. Three

deadlocks on the same two tables every hour is a design problem. Bob parses these graphs on a 5-minute cycle, correlates them against application activity patterns, and flags recurring cycles for investigation.

MCP tool. A capability registered with an agent through the Model Context Protocol. When Bob's reasoning layer decides it needs to run a diagnostic query, scan the network, or send an alert, it calls a named tool with typed parameters. The tool does the actual work and returns a structured result. Tools are the hands. The LLM is the brain deciding when to reach for them. Bob has 24 inline tools (including `run_command`, `network_scan`, `docker_action`, `mcp_call`, `write_file`, and `notify`) and 210 additional tools loaded from feature modules at startup. The `mcp_call` tool is the gateway to 43 registered MCP servers covering databases, infrastructure, network management, and more.

System prompt. The instructions given to an LLM before any conversation or tool message. It defines the model's role, its constraints, what it is allowed to do, and what it must never do without preconditions met. Bob's ReAct system prompt tells the model it is Ward's executive assistant and master IT/cybersecurity engineer, lists every available tool with descriptions, includes a dynamically generated summary of all 43 MCP server capabilities grouped by category, and specifies the mandatory human-in-the-loop rule: before executing any action that changes network, OS, filesystem, services, or security, check authority first and use `create_task` to queue it on the Kanban board for Ward's approval. It is the most important configuration file in the stack. A precise system prompt produces precise output. An ambiguous one produces creative interpretations you did not want.

Back Matter

Acknowledgments

The Navy taught me that systems do not maintain themselves. Someone has to care enough to be awake at 3 AM with a flashlight and a checklist. That discipline transferred directly to database work, and that discipline is under everything Bob does.

Every DBA team I have worked with over twenty years contributed something to the diagnostic patterns in this book. The teams at Chickasaw Nation, TWIA, Lumeris, Entergy (through ComTec), and Dogsbarking each dealt with failure modes I had not encountered before, and each one is somewhere in the wait stats filter list or the system prompt templates. You taught me what to look for by making me look.

The AI-for-DBAs community is still early. The people who have engaged with the early versions of these ideas, challenged the prompt templates, pushed back on the auto-apply scope, and shared their own failure cases: you are the reason

the prompt templates and the auto-apply scope have been stress-tested by more than one operator.

Get Involved

The work in this book is one DBA's home lab. The community is how it scales.

If reading this made you think *I could run this against my own instances*, there are three places to go from here.

Join the AI for DBAs community

The prompt templates, the failure cases, the pattern library, and the conversation about what to build next all live in two places:

- **LinkedIn newsletter:** Subscribe on LinkedIn
- **Skool group:** ai-for-dbas-7678

Both are free. Both are where I post diagnostic recipes, system prompt revisions, and the war stories the book is too short to fit. Bring your own.

Send your war stories and ideas

A finding the agent missed. An extension you want to build. A wait-stats pattern that does not show up here. A configuration that makes Bob safer for your environment. Email me at wardminson@minsondata.com.

I read every message. Failures are as useful as wins. The next version of this book will include the patterns that come back from readers actually running this in production.

If you are an investor

I am actively looking for backing to take Bob from a home lab project to a productized agent for managed-services providers and regulated-industry data teams. The architecture is proven. The stack is open. The compliance story is real. The market is every DBA who cannot send their query plans to a cloud API.

If you want to talk about funding the next stage, email wardminson@minso ndata.com with subject line `Bob investor inquiry`. I will send the technical due-diligence package and we can find time to talk.

Bibliography and Source Materials

The technical claims in this book are grounded in production documentation that lives alongside the manuscript in source control:

- **Infrastructure truth:** `/home/ward/Projects/01-PRODUCTION/new-network/new-docs/NETWORK_DOCU` Authoritative host inventory, IP assignments, open ports, and service dependencies as of the manuscript date.
- **Bob agent source:** `/home/ward/Projects/08-MY-AGENTS/Bob/docs`. The production agent documentation including full system prompt history, probe query versions, and deployment runbooks.
- **SQL Server DMV references:** Microsoft SQL Server documentation, `sys.dm_os_wait_stats`, `sys.dm_exec_query_stats`, `sys.dm_exec_query_plan`, `sys.dm_db_index_physical_stats`. All DMV queries in this book run against SQL Server 2019+.
- **MCP specification:** Model Context Protocol specification, Anthropic, 2024. The MCP integration in Bob implements the tool-calling interface as specified.
- **Proxmox VE documentation:** Proxmox VE 8.4.1 API reference. The snapshot and rollback calls in Chapter 8 use the REST API at `/api2/json/`.

Colophon

This manuscript was written in Markdown and version-controlled in Gitea at `http://10.0.0.80:3000/hyp3rsoft/the-birth-of-bob`.

Build toolchain:

- Markdown source: one file per chapter, YAML front-matter per `TestPlan.md §2`
- Diagrams: Mermaid, rendered to SVG by `mermaid-cli` during the build
- Lint: `markdownlint-cli` and `vale`
- PDF export: Pandoc 3.x with LaTeX
- EPUB export: Pandoc 3.x
- Static site: MkDocs Material with Glassmorphism CSS overrides
- CI: Gitea Actions on `gitea-runner` at `10.0.0.80`
- AI assist for draft passes: local Ollama on `inference-host-1` at `10.0.0.70`, `gemma4:26b` for prose and reasoning, `qwen2.5-14b` on `inference-host-2` at `10.0.0.71` for technical sections

The AI-density gate requires every chapter to score below 15% on `ai-writing-patterns` before release. The voice-check gate requires every chapter to match the `ward-voice` profile. Both run as part of the CI pipeline on every commit to `main`.

All code samples carry a `// source: bob@c549c88` comment referencing the production Bob commit at the time of v0.1.0-rc1. Readers wanting the exact file each sample was adapted from can check out the Bob source at that commit and grep for the constant or function name shown in the snippet.

The Birth of Bob

An Autonomous Agent for SQL Server Self-Healing and Network Monitoring

Ward T. Minson

A book about building a local AI agent for regulated database environments, why cloud LLMs are wrong for the work, and what it took to put real autonomy in front of a production SQL Server.

Read

Start with the Front Matter and the agent vocabulary glossary, or jump straight to a chapter:

Download

- PDF, 263 KB, 69 pages
- EPUB, 91 KB

About

Source under `hyp3rsoft/the-birth-of-bob` on Gitea. Manuscript license: CC BY-NC-SA 4.0. Code samples: MIT.

Chapter 1: The Report Problem

The report was 214 pages long.

I had not set out to build something that large. It grew the way most technical debt grows: incrementally, each addition justified at the time.

I had the PDF open on one monitor and SQL Server Management Studio on the other. It was a Tuesday in March, some year where the word "digital transformation" was still being used unironically in executive slide decks. The report was the weekly SQL Server health check for a mid-size financial services client. I had written the script that generated it. I had scheduled the job. I had configured the SMTP relay so it landed in fourteen inboxes every Monday morning.

Nobody read it.

I knew this because I had buried a joke on page 87, something about a cat photo and a deadlock, and not a single person said anything. Page 87 was probably where most readers gave up, assuming they made it past the table of contents at all.

That is the report problem. Not the content. The content was fine. The content was accurate, well-structured, timestamped, cross-referenced. The problem was that the format had grown to match the anxiety of the person generating it. Every time I missed something in a previous report and an incident happened, I added a new section. Missed an index fragmentation spike? Add a fragmentation section. Missed a growing tempdb? Add a tempdb section. Missed a suspicious query regression? Add a top-query-by-CPU section. Do that for two years and you have a 214-page document that nobody opens until something breaks.

At which point the report is useless anyway, because you are in incident mode and you do not have time to read 214 pages.

The DBA's job is variance detection. Is this server behaving differently today than it was last Tuesday? If yes, why? If the why is benign, note it and move on. If the why is a problem, fix it or escalate it before anyone downstream notices.

The report was supposed to serve variance detection. It failed at this for two reasons.

The first reason is volume. 214 pages of data is not information. It is noise with

structure. Humans are not good at reading 214 pages of time-series charts and synthesizing the signal. We habituate. The eye moves over the page looking for something that looks wrong, and when nothing jumps out immediately, we mark the email read and go to our next meeting. The report was doing the cognitive work of data collection but not the cognitive work of interpretation. That interpretation work was still landing on the DBA.

The second reason is timing. The report ran at 6 AM Monday. By the time I read it Tuesday morning, the data was thirty hours old. If something was quietly getting worse across the weekend, I was not going to catch it at 6 AM on a Monday. I was going to catch it at 2 PM on a Tuesday when an application team called saying their batch job had been timing out since Sunday.

This is a solvable problem. It just requires thinking about it as a software problem rather than a documentation problem.

SQL Server exposes a remarkable amount of diagnostic information in real time. Dynamic Management Views, or DMVs, let you query the engine's internal state the same way you query a table. You can ask the server what its longest-running queries are. You can ask what its wait stats look like. You can ask what queries have changed execution plans since last week. You can ask whether any of your indexes have not been used in thirty days and are therefore dead weight.

None of this requires a scheduled batch job. You can ask these questions continuously.

The old approach was: run a batch job, collect data, format a report, email it to fourteen people, hope someone reads it. The new approach is: observe continuously, detect variance, and act on the variance immediately. The difference between the two is roughly the difference between weather forecasting via newspaper (read yesterday's weather report this morning) and weather forecasting via radar (see what's happening right now).

The problem with continuous monitoring is not technical. The problem is the interpretation layer. You can instrument everything. SQL Server will happily export metrics to a table, to a flat file, to a monitoring agent, to a Prometheus scrape endpoint if you set one up. But interpreting that stream in real time, correlating wait stats with query plans across application activity windows, still requires someone who understands what the numbers mean.

Historically that someone was a DBA. A human, reachable by phone, whose job it was to know which signals matter and which are noise.

That still works at a certain scale. It stops working when you have more SQL Server instances than DBAs, which is most organizations over a certain size. It stops working at 2 AM. It stops working when your experienced DBA leaves and takes fifteen years of pattern-matching knowledge with them.

I want to be precise about what I was frustrated with, because the frustration is the seed of Bob.

I was not frustrated with SQL Server. SQL Server is a well-built piece of software with genuinely good diagnostic tooling. I was not frustrated with monitoring in general. I was not even frustrated with the report per se.

I was frustrated with the gap between the diagnostic information that already existed in the system and the humans-in-the-loop required to interpret it. The information was there. The system knew it had a problem. The problem was sitting in a DMV somewhere, expressible in four lines of T-SQL. What was missing was something that could read that DMV, understand what it meant, and say something useful about it without me having to be awake.

That is a reasoning problem. Data collection was solved. Alerting thresholds were solved, badly, but solved. The missing piece was the ability to look at a cluster of signals, a query regression here, a wait stat anomaly there, an index usage change somewhere else, and synthesize a diagnostic that would make sense to a senior DBA reading it cold.

In 2022 you could not do that automatically. You could write rule engines, expert systems, heuristics. But those systems were brittle. They knew what you taught them. They could not generalize from a pattern they had not seen before. Every new failure mode required a new rule.

In 2024 you could do it differently.

Large language models turned out to be surprisingly good at understanding SQL Server diagnostic data. Not because they were trained on SQL Server specifically, though they were exposed to plenty of SQL and plenty of SQL Server documentation. The reason was more general than that. A well-trained model could read a query plan XML, understand what the shapes of slow plans look like, and generate a plausible diagnostic hypothesis. It could read a wait stats snapshot, understand what `CXPACKET` waits mean at high volume, and suggest whether you were looking at a parallelism problem or a legitimate parallel scan on a large table.

It was not magic. It was wrong sometimes. It hallucinated occasionally. The first few months of working with LLMs on SQL diagnostic problems were an education in knowing when to trust the output and when to verify it independently.

But it was dramatically better than a rule engine. It generalized. It could reason about a combination of signals it had never seen in exactly that configuration before. It could explain its reasoning in plain English that a non-DBA developer or a manager could read and act on without a translation layer.

That was the insight. AI would not replace DBAs at the level of actual system design and incident response, but the interpretation gap, the distance between

raw diagnostic data and actionable human language, was exactly the kind of gap a language model could bridge.

The question was how to build that bridge without handing your SQL Server's internal state to an external API.

That question is why Bob exists. And it is why Bob runs locally on a machine I own, on a network I control, using models that have never heard of your query plans and never will tell anyone about them.

There is a version of this story where I say: I saw AI coming and I was ahead of the curve. That version is false.

The honest version is: I had a problem that annoyed me for years, I kept a mental note of it, and then a technology arrived that looked like it might fit the hole. I did not move fast. I spent about six months reading about transformer architectures and local LLM options before I wrote a single line of code in that direction. I ran Ollama locally on a laptop first, queried it by hand with SQL fragments I copied and pasted, and watched whether the responses were useful.

They were useful enough. Not production-ready. Not reliable enough to act on without review. But useful enough to know the direction was right.

The cloud question came up immediately. OpenAI's API was obviously capable. Claude was available. I was using both at work, at Dogsbarking.com, for semantic mapping and entity resolution in Salesforce deduplication pipelines. GPT-4 and Claude 3 were handling complex reasoning tasks across large data sets, and doing it reliably enough that I was leaning on them for production work.

So why not use the cloud for Bob?

The answer was not ideological. It was the client list. The SQL Server environments I was thinking about when I designed Bob were not toy databases. They were production systems in industries where data governance is not optional: financial services, healthcare, utilities. At Lumeris in 2017 I was managing PHI under HIPAA. At Entergy through 2024 I was maintaining NERC CIP compliance. If I sent a query plan from one of those environments to an external API, I would need to explain to a compliance officer exactly which data left the network, under which data processing agreement, with which retention policy.

That conversation is possible but slow, and every organization that wants Bob to help them needs to go through a vendor review process for each AI provider in the chain. Cloud LLMs are not self-certifying for HIPAA or NERC CIP. You need a business associate agreement, a data processing addendum, or a vendor risk assessment before you can say that the AI assistant in your monitoring stack is compliant.

Local inference skips all of that. The data stays inside the network. There is no vendor. There is no data processing agreement to negotiate. There is nothing to include in your audit documentation beyond "we process this data on our own hardware using open-weight models." Your CISO can sign off on that sentence without a six-month review cycle.

That was the practical argument for local inference. It did not require me to have strong feelings about cloud AI in general. I use cloud AI. I use it at work when the compliance constraints permit it. The point is that "when compliance constraints permit it" is a significant qualifier for the environments where monitoring AI is most valuable.

Building Bob took eighteen months to reach its current form. Not eighteen months of continuous full-time work, but eighteen months from the first working prototype to the production system documented in Part 2. That time includes multiple architectural restarts, a period of six months where the project sat idle because other work took priority, and the slow accumulation of operational track record that is the only way to know whether the reasoning layer is actually trustworthy.

That duration matters because it sets an honest expectation. The architecture in Part 2 is not the first design. It is the design that survived contact with real operational conditions. The first design did not have a rollback mechanism. The second design had a rollback mechanism but no confidence gate on the actuator. The third design, which is roughly what is documented here, has both. Each iteration was motivated by something going wrong in a way that the previous design could not handle gracefully.

Chapter 2 covers what that development process looked like from the inside: the shift from running queries manually to building a system that could do it without me, and what the term "vibe coding" actually means when you are writing software that touches production data.

What I want to leave you with from this chapter is the problem statement, clearly. Because if you do not feel the problem, the solution is not going to make sense.

The report was 214 pages long. Nobody read it. A server would get sick, and then sicker, and then something would break, and then I would open the report and find the signals had been there for days. Sitting in a PDF. Waiting.

That is the report problem. That is what Bob is for.

Chapter 2: From DBA to Vibe Coder

I do not like the term "vibe coding."

I used it in the chapter title because it is currently the shortest path between the words and the concept. But let me tell you what I actually mean by it, because the term as it circulates on tech Twitter carries a connotation I want to separate from what I am describing.

The popular definition is something like: describe your feature to an AI in plain English, accept whatever it produces, ship it. Do not read the code. Do not understand the code. Just vibe.

That is not what I mean. That is the path to production software that nobody on the team understands, and when it breaks at 2 AM, it stays broken longer because the codebase is alien to everyone touching it.

What I mean by vibe coding, at least in the context of building Bob, is closer to this: focus on intent, not syntax. Know what you want the system to do. Be precise about the requirements and the constraints. Use AI to generate the structural code, the boilerplate, the framework scaffolding, because that work is low-value and time-consuming. But review everything. Understand everything. Treat the AI as a fast junior developer who needs close supervision, not as a senior architect whose output you trust without reading.

That distinction matters because Bob does things with real consequences. It runs T-SQL against production databases. It takes Proxmox snapshots. It sends alerts. An agent that does any of those things based on code nobody reviewed is a liability, not a feature.

The shift from pure DBA to something you could call a systems builder happened over a couple of years and was driven by necessity more than ambition.

The practical reality of working at a digital agency is that the DBA role does not exist in isolation. You are also the person who configures the monitoring server, writes the deployment scripts, sets up the Docker environments, and figures out why the staging database disagrees with production. Over time, the line between "database work" and "systems work" becomes irrelevant. You do what needs doing.

That context meant I already had a Python environment and some exposure to REST APIs before I started thinking about Bob. I was not coming in cold. But I was also not a developer. My code before AI assist was functional and ugly: lots of string concatenation, no type hints, test coverage as an afterthought. It worked. It was not something I would show anyone.

When I started using AI for code generation, two things happened. First, the output was better than my equivalent unassisted output, at least structurally. The AI wrote cleaner function signatures, used the right data structures, caught edge cases I would have missed. Second, and this is the part I did not expect: I started learning faster. Not because the AI explained things, though it did when I asked. Because I was reading more code in a day than I had read in

a week before. You learn to code by reading code. If AI can generate twenty good examples of a pattern in the time it takes me to write one bad one, that is a faster feedback loop.

The combination of those two things meant that within about six months, my ability to build systems had expanded meaningfully. Not because I was a better programmer. Because the cost of attempting something had dropped. I would try an approach, read what the AI gave me, modify it, test it, and discard it if it did not work. The discard rate was high. That was fine.

The first Bob prototype was not called Bob. It was called something embarrassing that I am not going to put in this book.

It was a Python script, about 200 lines, that connected to SQL Server via `pyodbc`, ran a handful of DMV queries, bundled the results into a prompt, and posted them to an Ollama API endpoint. The response came back as plain text. I printed it to the console.

That was it. That was the prototype.

Let me describe one specific session that was representative of how the thing got built, because the retrospective version sounds cleaner than the reality was.

The feature I was trying to add was deadlock detection. I knew what I wanted: the agent should extract deadlock events from the SQL Server system health session, parse the victim chain, and include a summary in the diagnostic context. I knew the T-SQL to query the system health extended event session. What I did not know was how to parse the deadlock XML in Python in a way that handled the multiple schema variations SQL Server uses depending on the version and the deadlock type.

I described the problem to the model. Not "write me a deadlock parser." That gives you generic code that handles none of the specific SQL Server quirks. I described it as: "I need to extract the victim process ID and the blocking resource from SQL Server deadlock XML that looks like this," and I pasted an actual deadlock XML blob from my test environment. The model gave me a parsing function using `xml.etree.ElementTree` that worked for that specific deadlock shape. I ran it. It worked on the example. I tested it against three other deadlock XML samples from the system health session. Two of them had a different structure for the `blockingObject` node. The parser broke on those.

I fed the failing examples back and asked it to handle both structures. It revised the function. I tested again. Still failing on one case, a deadlock involving a row lock on a heap table with no clustered index, which has a different XML structure than a deadlock on an indexed table. Third iteration, I described the specific difference. The model produced a function that handled all four cases I had tested. I added a `try/except` wrapper around the whole thing to handle

any fifth case I had not seen yet, logged the raw XML when the parser failed, and shipped it.

That session took about 45 minutes. Writing the same parser from first principles, reading the XML schema documentation, handling all the edge cases myself: call it three hours minimum, and I would have been less confident in the result because my XML handling in Python was not strong. The model's XML handling was better than mine. I reviewed every function it produced. I tested it against real data. I made the judgment calls about what "good enough" looked like. The collaboration produced something I understood and trusted.

That is vibe coding done right. Not accepting the first output. Using the model to close the gap between "I know what I want" and "I know how to build it," then applying your own judgment to decide when you got there.

The response quality was inconsistent. Sometimes the model said useful things. Sometimes it said things that were technically accurate but irrelevant to the actual data. Sometimes it hallucinated column names from DMVs that do not exist. I tuned the prompt, read about temperature settings, tried different models.

The skepticism I had about this process did not go away quickly. Twenty years of building muscle memory around SSMS, query plans, and DMV queries is a real thing. The reflex when something breaks in a database is to open a familiar tool, run a familiar diagnostic, apply a familiar fix. There is speed in that reflex. There is also a ceiling: you can only move as fast as your hands can type the queries you already know.

What shifted was a specific moment, not a gradual conversion. I was investigating a slow query that had suddenly gotten worse after a deployment, standard scenario. The developer who deployed the change swore the stored procedure was identical. The query plan had regressed from a seek to a scan on a key table. I went through my usual process: checked the statistics date, ran the update, compiled new plans, tested. Still scanning. Spent about an hour on it before I was willing to admit I was not seeing something.

I described the situation to the model, pasted the before and after query plans, and asked what changed. It came back in about fifteen seconds with: the parameter sniffing issue is on this specific parameter, the plan was compiled when the parameter value was low-cardinality, and the deployed change probably altered the stored procedure with `RECOMPILE` removed. Check the procedure definition for the `WITH RECOMPILE` option.

That was it. The developer had changed the procedure signature as a "minor cleanup" and removed an option that I had added years earlier and that had no comment explaining why it was there. The model spotted the structural difference in the plans that I had been staring at for an hour. Not because it was smarter than me about SQL Server. Because it was looking at the data fresh without the hour of sunk cost that made me keep reaching for solutions I

had already tried.

That was the conversion. Not "this is better than me." Just: sometimes the fresh eyes are the tool you need, and if the fresh eyes can do it in fifteen seconds at any hour, that changes what you build around them.

Over the course of several weeks, the consistency improved. Not because the model got better, though I was switching between versions and sizes. Because the prompt got better. I learned what information the model needed in context to reason well about SQL Server state. A raw wait stats snapshot with no baseline context produced worse output than a snapshot plus a rolling average plus a note about what the application load was doing. The model was not magic. It needed good inputs.

That prompt-engineering process was the first real vibe-coding lesson. You are not writing code when you write a system prompt. You are writing a contract. You are specifying what the agent knows, what it is supposed to do, and what constraints it must respect. Clarity in the contract produces clarity in the output; ambiguity produces creative interpretations you did not want.

The prototype validated the direction but exposed the limits of a 200-line script.

The first limit was state. A single prompt-and-response exchange has no memory. If the model identified a suspicious query at 9 AM and I wanted to check whether it was still suspicious at 2 PM, I had to start from scratch. There was no continuity, no ability to say "earlier you flagged this query as a candidate for index optimization; has the wait pattern changed?"

The second limit was action. The prototype could diagnose. It could not do anything about what it diagnosed. The output was text. Someone, me, had to read that text and decide whether to act. For after-hours monitoring, that defeats the purpose.

The third limit was integration. The prototype knew about SQL Server. It knew nothing else. If the cause of a query performance problem was a VM on the hypervisor starving for RAM, the prototype could not see that. It only saw SQL Server state.

Addressing all three limits was the transition from "script that wraps an LLM" to "agent architecture." That architecture has a name now: three layers, two inference hosts, one Model Context Protocol server tying it together. Part 2 documents it in full.

But the shift in thinking that made the architecture possible was simpler than the architecture itself. I stopped asking "how do I make the AI smarter" and started asking "what does the AI need to be useful?"

It needed memory. It needed tools. It needed a scope it could reason about completely, not just a fragment of the system state. Once I framed the problem

as "give the model what it needs to reason well and give it hands to act with," the architecture followed from those requirements pretty directly.

The word "vibe" in "vibe coding" captures something real, even if the full phrase is a little glib. When you work this way, there is a mode shift compared to traditional programming. You are not sitting down with a blank file and building upward from first principles. You are having a conversation with the system about what you want. The conversation has a feel to it. You know when you are getting close. You know when the output is technically correct but missing the point.

That is not a replacement for knowing your domain. It is the opposite. The better you know SQL Server, the better your prompts get. The better you understand agent architectures, the better you can evaluate what the AI generates. Vibe coding, done well, amplifies domain expertise. It does not substitute for it.

The DBA background was not a liability in this process. It was the whole point. I knew what questions to ask. I knew what a good diagnostic read like. I knew which signals to weight and which were noise. The AI handled the code generation. I handled the judgment.

The next chapter talks about why "something similar" matters at the organizational and community level. But before we go there: in Chapter 4, the hardware is real, the IPs are production, and the machine learning is running in your basement, not someone else's data center.

Chapter 3: AI for Everyone

There is a version of the Bob story that stops at "I built a tool for my own use." That version is fine. The tool works. It saves time. End of story.

But that is not the story I want to tell, because I do not think the problem Bob solves is mine alone.

The DBA profession has a specific skill distribution problem. The senior people, the ones who have been reading query plans since before the internet was fast enough to stream video, are expensive, overextended, and in most organizations, on a slow path to retirement. The junior people have energy and can write T-SQL, but they lack the years of exposure that turns a wait stats snapshot into a diagnosis. The middle layer, someone with five to ten years who can read a query plan and knows what `SOS_SCHEDULER_YIELD` waits mean at high volume, is genuinely rare.

This means most organizations are under-resourced on SQL Server monitoring.

The senior DBA is spread across too many instances. The alerts fire too often or not enough, because tuning alert thresholds requires the same expertise that's in short supply. And when the junior person gets paged at 3 AM about a high-CPU event, they open the monitoring dashboard, see a lot of red, and call the senior DBA anyway.

That pattern is not sustainable. It is also not a money problem. You can throw budget at it by buying managed monitoring services, and plenty of organizations do. But managed services have their own problems: data leaves your environment, the service has to be generic enough to work for thousands of customers, and the recommendations it generates are templated. They do not know your application. They do not know your data access patterns. They know what slow queries look like in general.

Bob knows your specific instance. It runs in your environment. It has been watching your specific workload. That is a different kind of intelligence.

The "AI for Everyone" claim is specific: the interpretation gap I described in Chapter 1 is solvable for organizations that cannot afford to staff it with senior DBAs.

Regulated-industry DBAs have the most to gain here and have been the most systematically left out of the AI monitoring conversation. A DBA at a hospital network, an electric utility, a regional bank: these are the people with the hardest monitoring problems, the highest stakes for getting it wrong, and the fewest options for bringing in cloud AI to help. HIPAA covers protected health information. NERC CIP covers critical infrastructure data. PCI DSS covers cardholder data. None of those standards were written with "send your diagnostic data to a third-party inference API" as an approved control. They were written before that was a real question.

Local inference does not solve the regulatory compliance problem automatically. You still need to document your data processing, maintain access controls, and audit what the agent is doing. But it eliminates the third-party data processor problem entirely, which removes the most difficult compliance question from the table. A CISO who needs to sign off on a DBA using an AI monitoring tool can say yes to "runs on our hardware, data never leaves our LAN" without a six-month vendor review. That changes which organizations can use the tool.

The math is this. A local LLM running on a reasonably modern GPU costs roughly $30 to $80 per month in electricity. A cloud API for the same inference volume costs hundreds to low thousands of dollars per month, and more importantly, it means your query plans, your wait stats, your application-specific diagnostic data, leaves your network. For a financial services client, a healthcare organization, any entity under data governance pressure, that is not a tradeoff you can make casually.

A dedicated machine running Ollama with a model capable of SQL Server reasoning costs somewhere between $500 and $3,000 depending on your ambition, consumes a few hundred watts, and pays for itself in the first avoided incident. That calculation changes the accessibility of serious monitoring AI dramatically.

This is the sovereignty argument, and Part 2 of this book makes it at length. But the origin of the argument is practical, not political. I did not start from "I don't want to pay OpenAI" as a philosophical position. I started from "my clients have regulated data and cannot use external APIs for this," and worked backward to the local inference setup; the philosophical position came after the technical requirement.

The moment I decided this was worth building for more than myself was not dramatic. It was a phone call.

A DBA I had worked with a few years earlier was getting paged every third night because his company's monitoring tool was generating false positives he could not tune down without also missing the real ones. He knew what to look for. He had been doing this for fifteen years. The problem was not expertise. The problem was that his expertise was not available at 2 AM when he was asleep and the alert fired. He would wake up, look at the same wait stats he had looked at a hundred times, determine it was the same network IO pattern that always resolved itself in ten minutes, go back to sleep. Then thirty minutes later it would fire again because the monitoring tool had no memory of the first alert.

That conversation was the one that clarified the mission. The interpretation gap was not just my problem. It was a profession-wide problem. And the solution was not more training, because the people getting paged at 2 AM already knew their databases. The solution was a system that could hold the expertise of the senior DBA and apply it continuously, without needing to wake anyone up.

The line on the resume, "AI for DBAs: Creator and strategist for communities focused on the future of data management," is where that mission sits publicly. The vision behind it is narrower and more specific than the phrase suggests: get enough senior DBAs to encode their diagnostic reasoning into systems like Bob that the knowledge stops being locked inside individual people's heads.

The hardest people to reach with that argument are the DBAs who are most qualified to act on it. Senior DBAs tend to be skeptical about automation for good reasons. They have seen automation fail in expensive ways. They have cleaned up after systems that were confident and wrong. They carry a healthy distrust of anything that promises to replace judgment with code. That distrust is correct in general, and the argument Bob makes is not that code replaces judgment. The argument is that your judgment, encoded as structured instruction and system prompt context, can work while you sleep. The code is just the delivery mechanism for expertise you already have.

"AI for Everyone" also has a community dimension.

DBAs who build their own tooling tend to share it poorly. The culture of the DBA community historically has been generous with knowledge, SQL Server forums have always been active and helpful, but stingy with code. Sharing a diagnostic query is common. Sharing a full monitoring agent with deployment instructions is rare.

Part of that is legitimate IP concern. If you build something that gives your organization a competitive advantage, you do not necessarily want to open-source it. Part of it is simpler: sharing code requires documentation, and documentation takes time, and DBAs are usually busy.

Bob is meant to be a template, not just a product. The architecture documented in Part 2 is intentionally separable from my specific infrastructure. The three-layer design works for a single SQL Server instance at a small organization or a fleet of instances at an enterprise. The Ollama integration is not tied to the specific model I run. The MCP tool set is extensible by design.

The vision is a community of DBAs who have adapted the architecture to their own environments, contributed new tools, stress-tested the system against failure modes I have not encountered, and generally made it better than I could alone. The hard part of that vision is not technical. It is social. Getting DBAs to share working code, to trust that what someone else built is actually useful and not a trap, takes trust that has to be built over time.

That is an ongoing project. Chapter 9 talks about the community side in more detail.

There is a last thing worth naming in the origin section, before Part 2 gets into the machines and the protocols.

Building Bob changed how I think about the role of experience in technical work.

I spent a lot of years thinking that the value I added as a DBA was primarily in the form of knowledge. I know what this wait stat means. I know what this query plan shape indicates. I know which configuration knobs to reach for when memory pressure looks like this. That knowledge was hard-won and genuinely useful.

What I learned building Bob is that the value is also in the questions. What to ask. Which signals to correlate. When to look at the infrastructure layer instead of the query layer. When to escalate instead of fix. When to leave it alone because the apparent problem is the system responding correctly to unusual load.

Those questions are not encoded in DMV documentation. They come from experience. And they can, to a meaningful degree, be encoded in a system

prompt. You can write down what you know, what you look for, what your diagnostic decision tree is. When you do that, the AI has your framework to reason within. You are not asking it to invent DBA expertise. You are giving it yours, and asking it to apply it faster than you can.

The tool works because a senior DBA built it. It will be useful to someone who isn't a senior DBA because the senior DBA's reasoning is baked into it. That is the democratization story: not "anyone can use AI to do DBA work," but "a DBA can capture their expertise in a form that works when they're asleep."

The hardware for that system lives on a server in my home lab, and it is more capable than the price suggests. Chapter 4 walks you through it in detail.

Chapter 4: The Infrastructure

Bob does not run in the cloud, by design. The off-cloud placement is the architecture, not a limitation to work around.

This chapter documents the physical infrastructure Bob runs on: the hardware, the hypervisor, the network, the storage. Every IP address in this chapter is real. Every hostname is production. If something in this chapter contradicts what you see when you SSH into the relevant host, the chapter is wrong and should be corrected.

The Network

The lab runs on a single `/24` subnet: `10.0.0.0/24`. The router is OPNsense at `.1`, running FreeBSD with Unbound for DNS and a full stateful firewall. DHCP is centralized on OPNsense. DNS resolution is handled by two BIND 9.20.11 forward-only resolvers: `dns-primary` at `.222` and `dns-secondary` at `.221`, both running as Proxmox VMs.

The primary hypervisor is Proxmox VE 8.4.1 at `10.0.0.2`. It runs on an i9-13900K with 94 GB of RAM and is the host for most of the services Bob depends on. The VM inventory relevant to Bob is listed below.

NAS storage lives on `nas` at `10.0.0.20`: Ubuntu 25.10 with Samba acting as an Active Directory domain controller, four NICs bonded into `bond0` for throughput, and 37 TB of RAID for bulk storage.

The SQL Server target Bob monitors runs on `db-host` at `10.0.0.14`. This is a Windows Server 2012 host with SQL Server listening on the non-standard port `50003`. The `BookOfBob` metadata database lives here alongside the production databases Bob monitors.

OPNsense at `.1` handles more than routing. It runs Unbound DNS, which means internal resolution is fast and fully under local control. No DNS query for an

internal hostname leaves the LAN. DHCP leases are managed centrally here too, which matters for monitoring: static assignments for every host the agent touches, since an IP address change at 3 AM because a DHCP lease expired is exactly the kind of silent failure that produces a 4 AM page.

The two BIND resolvers at `.222` and `.221` are forward-only. They receive DNS queries from LAN hosts, forward to Unbound on OPNsense or upstream resolvers, and cache results. Redundant DNS is infrastructure hygiene. If the agent host cannot resolve `db-host.local` because a single resolver VM is restarting, Bob goes blind. Two resolvers cost four vCPUs and 8 GB of RAM on Proxmox. That is a cheap insurance premium.

The AI Hosts

Bob's inference layer runs on two dedicated AI servers.

`inference-host-1` at `10.0.0.70` is the primary inference host. Hardware: Ryzen 9 9950X processor, RTX 3090 GPU with 24 GB VRAM, 182 GB system RAM. This machine runs Ubuntu 25.10 with Ollama managed as a systemd service, Open WebUI for browser-based model interaction, and two text-to-speech containers. The RTX 3090 is critical here: 24 GB of VRAM is enough to run `gemma4:26b`, Bob's primary model, without falling back to CPU inference. CPU fallback on a model this size is usable for experimentation and painful for production monitoring.

`inference-host-2` at `10.0.0.71` is the secondary inference host. Hardware: i7-11700F, RTX 3060 with 12 GB VRAM, 125 GB system RAM, Ubuntu 24.04.4. Ollama runs as a systemd service here with fifteen models available, including `qwen2.5-14b` which serves as Bob's automatic failover model. The RTX 3060 handles models up to about 13B parameters comfortably at Q4 quantization. Larger models require careful quantization choices to fit in 12 GB.

Both hosts expose Ollama at the standard port `11434`. Bob's `_ollama_call()` function implements automatic failover: if the primary host at `.70` is unreachable, it transparently tries the next entry in the `llm_config.json` failover chain, which points to `qwen2.5-14b` on `.71`. Each server has its own default model since different servers host different model sets. No human intervention is needed for a failover event.

The Ryzen 9 9950X in `inference-host-1` is worth a note because the CPU choice matters more than it seems. Ollama offloads model layers to GPU VRAM as much as possible, but when a model's total parameter footprint exceeds available VRAM, the overflow runs on CPU. For `gemma4:26b`, Bob's primary model, the model fits in 24 GB VRAM with room to spare, so CPU inference is not a normal-path concern. Where the CPU matters is in tokenization, in the embedding generation for the context window, and in parallel requests: when two monitoring cycles happen to coincide, the second one does not wait for the

first to complete GPU inference if the CPU can handle the non-GPU work in parallel. The 9950X's 32 threads handle that without contention. An older eight-core machine would not.

`inference-host-2` at `.71` tells a different story. The i7-11700F is a capable processor but it predates the efficiency cores that make the 9950X fast at mixed workloads. The RTX 3060 with 12 GB VRAM is the binding constraint: `gemma4:26b` does not fit. The failover model `qwen2.5-14b` at Q4 quantization uses about 9 GB of VRAM, leaving 3 GB for context and system overhead. That 3 GB margin is tight. Running concurrent requests on the 3060 with the 14B model loaded will cause model layers to spill to system RAM, and the performance penalty is significant enough that in practice, the fallback host should not be expected to match `inference-host-1`'s throughput or response latency.

That asymmetry is fine for a failover scenario. Bob running on `inference-host-2` is slower and less capable than Bob running on `inference-host-1`, but it is running. The alternative to a slower fallback is no monitoring when the primary is down.

The Hypervisor and Supporting VMs

Proxmox at `10.0.0.2` hosts the VMs that provide Bob's supporting infrastructure. The table below shows the VMs that matter to this deployment.

VM ID	Name	vCPUs	RAM	IP	Role
101	dns-primary	4	4 GB	.222	BIND 9 primary DNS
102	dns-secondary	4	4 GB	.221	BIND 9 secondary DNS
103	gitea	4	16 GB	.80	Source control, CI
104	webserver	8	32 GB	.50	Apache, web serving
106	n8n	4	8 GB	.60	Workflow automation

The Gitea VM at `10.0.0.80` is where Bob's source lives: `http://10.0.0.80:3000/hyp3rsoft/bob`. Gitea's SSH port is `2222`, not the standard `22`. The CI runner (`gitea-runner`) lives on the same VM and handles the build pipeline.

Bob agent VMs are provisioned on Proxmox as needed. The agent itself is lightweight enough to share a VM with other services in a lab context. For production deployments, dedicated VM allocation is recommended.

The `n8n` workflow automation VM at `.60` is not part of Bob's primary architecture, but it handles the glue work: webhook routing, notification formatting, and the scheduled tasks that do not belong in the agent loop. When Bob fires an alert, the destination is an n8n webhook endpoint. n8n decides whether that alert becomes an Asterisk call, a formatted message to a chat channel, or a ticket

in the issue tracker. Keeping that routing logic out of the agent code means it can be changed without deploying a new version of Bob.

The webserver VM at `.50` hosts the Bob documentation site, built from the manuscript's Markdown via the Gitea CI pipeline and served under Apache. That is not operational infrastructure for Bob, but it is part of the project. Cloudflare terminates SSL at the edge and proxies to the Apache vhost, which means the documentation is publicly accessible without exposing the LAN.

The Proxmox Decision

The hypervisor choice is Proxmox VE 8.4.1 running on the i9-13900K machine at `10.0.0.2`. That decision deserves an explanation because it is not the obvious choice and the alternatives are real.

The first alternative is bare-metal Docker: run everything as containers directly on the host operating system, no hypervisor. This is simpler initially. It is also the choice that produces the most pain at scale. Container-based isolation is not VM-based isolation. A runaway container process can consume resources that affect other containers on the same host. A hypervisor with defined vCPU and memory limits on each VM does not have that problem. For a home lab running one or two services, Docker on bare metal is fine. For a lab running eight VMs with different security boundaries and resource requirements, the hypervisor overhead is worth it.

The second alternative is ESXi. VMware's hypervisor is the enterprise standard for a reason: it is stable, well-documented, and most corporate IT departments already know it. The reason it is not the choice here is licensing. VMware's acquisition by Broadcom in 2024 changed the licensing model in ways that eliminated the free tier and made single-node home lab deployments economically unreasonable. Proxmox is open source, actively developed, and free for single nodes without a support contract.

The third alternative is k3s or another lightweight Kubernetes distribution. This is the forward-looking choice for anyone building microservices at scale, and if the project's trajectory was toward hundreds of small services, it would be the right call. Bob is not hundreds of small services. It is a handful of well-defined components that do not benefit from the orchestration overhead of Kubernetes. Using k3s here would be choosing the tool that will impress people reading your architecture diagram over the tool that is right for the problem.

Proxmox on a 94 GB, 32-thread machine gives the lab a vCPU overcommit ratio of 1.06x across 34 allocated vCPUs, barely overcommit. The VMs mostly idle, which means the ratio on active threads at any given moment is much lower. RAM is the tighter constraint: 77 GB allocated against 94 GB available. That leaves 17 GB for the hypervisor overhead and any burst. It is tight but

manageable because the AI inference hosts are dedicated hardware, not VMs, so the RAM-hungry inference workload never touches Proxmox's RAM budget.

The Network Topology

```
graph TD
    inet[Internet] --> cf[Cloudflare Tunnel]
    cf --> ws[webserver .50]

    opn[OPNsense .1] --> prox[Proxmox .2]
    opn --> nas[nas .20]
    opn --> ai1[inference-host-1 .70]
    opn --> ai2[inference-host-2 .71]
    opn --> win[db-host .14]

    prox --> gitea[Gitea VM .80]
    prox --> web[webserver VM .50]

    ai1 -- "Ollama :11434" --> bob[Bob Agent]
    win -- "SQL :50003" --> bob
    gitea -- "source control" --> bob

    bob --> win
    bob --> ai1
```

Sizing for Bob

The infrastructure above is a production setup at the high end of what you need for a single-instance SQL Server monitoring deployment. The table below gives honest numbers for three deployment sizes.

Tier	CPU	GPU / VRAM	RAM	Storage	Use Case
Minimum	8-core (any modern x86)	RTX 3060 / 12 GB	32 GB	500 GB SSD	Single SQL Server instance, models 13B, response time 5–30 s

Tier	CPU	GPU / VRAM	RAM	Storage	Use Case
Recommended	16-core (Ryzen 7 / i9)	RTX 3080/3090 / 16–24 GB	64 GB	1 TB NVMe	2–4 SQL Server instances, models up to 32B Q4, response time 2–8 s
Production	16-core+ (Ryzen 9 9950X)	RTX 3090 / 24 GB	128 GB+	2 TB NVMe RAID	5+ instances, concurrent monitoring, redundant inference host

A few notes on the numbers.

The VRAM figure is the constraint that matters most. System RAM can be large or small; Ollama uses system RAM as overflow when a model layer spills off the GPU, but at inference time you want as much as possible on the GPU. A model layer on CPU is roughly 10x slower than on GPU. For a monitoring agent that might need to respond within a few seconds of an alert firing, 10x slower means the difference between "useful" and "too late to matter."

The minimum tier is genuinely minimum. A single RTX 3060 with 12 GB VRAM will run `qwen2.5-14b` or a similarly-sized model at Q4 quantization with acceptable latency for batch monitoring tasks. It will struggle if you try to run a 26B+ model or run concurrent requests. For a self-hoster who wants to try this on a weekend, the minimum tier works. For production, reach for the recommended tier.

Storage matters less than you might expect for inference, because model weights are read once at load time and then the model lives in GPU/system RAM. The storage budget is mostly for the model files themselves, which range from about 4 GB for a 7B model at Q4 to about 20 GB for a 32B model at Q4. Keep 200 GB free for a comfortable model library with room to grow.

The "production" tier listed above maps to the actual `inference-host-1` hardware. It is not aspirational. It is what I am running.

The minimum tier deserves more attention because it is where most people starting out will land. An RTX 3060 with 12 GB VRAM and `qwen2.5-14b` or a comparable 14B model will work. The responses are slower and the diagnostic quality on complex query plans is lower than with the 26B model running on the production tier. The failure mode you will hit first at the minimum tier is not a crash or an error. It is the model returning a finding that misses the root cause of a complex problem. Wait stat correlation with query plan analysis is where smaller models struggle most. If your SQL Server workload

is straightforward, mostly OLTP, well-indexed, the 14B model is adequate. If you are running complex analytical workloads with large parallel queries and frequent plan regressions, the minimum tier will produce diagnostic output that sounds plausible but requires more human verification.

The recommended tier at 64 GB RAM matters for a reason separate from inference. The Proxmox snapshots that the remediation pattern depends on require disk space proportional to VM size. If your SQL Server VM has 16 GB RAM and a 200 GB disk, a pre-fix snapshot will consume roughly the changed-page delta, which is small for most index maintenance operations. But if you are running multiple agents against multiple VMs and each monitoring cycle produces a snapshot, disk consumption grows. 1 TB NVMe with fast write throughput keeps the snapshot operation fast enough that it does not become a bottleneck in the remediation flow.

The lab did not start at the production tier. At month zero, Bob ran on the development workstation at `10.0.0.100`, a desktop-class machine with an RTX GPU and enough RAM to run the 14B model. That was where the 200-line prototype lived. By month six, the prototype had grown into something that needed its own host, and `inference-host-2` at `.71` came online as the primary inference host. The RTX 3060 was enough to validate the architecture. By month twelve, the diagnostic quality limitations of the 14B model were clear enough that the case for dedicated hardware was easy to make. `inference-host-1` at `.70`, Ryzen 9 9950X, RTX 3090, 182 GB RAM, came online around month fourteen and has been the primary host since. The two months between twelve and fourteen were the procurement and setup time.

That arc, workstation to dedicated mid-range hardware to dedicated high-end hardware, mirrors the arc of what Bob could do. At month zero, it could diagnose. At month six, it could recommend. At month twelve, it could act in narrow, low-risk cases. By month eighteen, the auto-apply scope had expanded to cover the full index maintenance category, and the architecture was stable enough to document.

Network Security Notes

The setup described here is a LAN deployment with no exposure of inference endpoints to the internet. Ollama at `10.0.0.70:11434` is not firewalled from the internal LAN (all LAN hosts can reach it), which is an open finding from the NETWORK_SCAN.md audit. For an organization deploying Bob in a regulated environment, the recommendation is to firewall `11434` to permit only the specific hosts that need it: the agent host, the management workstation, and the Gitea CI runner.

The SQL Server host at `.14` listens on `50003`. The non-standard port is a defense-in-depth measure, not a security boundary. Proper SQL Server security

requires strong service account credentials, auditing enabled, and network access controlled at the firewall, not just the port number.

Authentication between Bob and the SQL Server uses a dedicated service account with the minimum permissions required: `VIEW SERVER STATE` for DMV access, `VIEW DATABASE STATE` on monitored databases, and write access to `BookOfBob` for logging. No `sysadmin` membership. This is important for the same reason it always is: the blast radius of a credential compromise should be as small as possible.

The MCP server that connects Bob's reasoning layer to its actuators uses authenticated tool calls. The anonymous bind finding from `NETWORK_SCAN.md` applies to the LDAP service on `nas`, not to the Bob toolchain, but it is worth noting: if you are building on this infrastructure, audit every service that accepts anonymous authentication and close those gaps before putting an agent on the network that can take action based on what it finds.

The Data Residency Guarantee

Every byte of diagnostic data that passes through Bob stays inside `10.0.0.0/24`.

Query plans, wait stats, deadlock graphs, T-SQL text: none of it leaves the LAN. The Ollama instance receives this data as part of the inference prompt and processes it entirely in memory on `inference-host-1`. There is no telemetry to external services. There is no model-training feedback loop that uploads your data somewhere. The models themselves are static weights that were downloaded once and run locally.

This is not difficult to verify. Run `tcpdump -i eth0 not src 10.0.0.0/24 and not dst 10.0.0.0/24` on the agent host during a monitoring cycle. You should see nothing. If you see outbound connections on the inference ports, something is misconfigured.

The data residency guarantee is the thing that makes Bob viable for regulated environments. It is also the thing that makes it interesting as an architecture. The system's intelligence is local. It requires good local hardware, which is cheap compared to the alternative, and good local models, which are free.

Chapter 5 documents the Ollama side of this in detail: the specific models, the configuration, the systemd setup, and the hot-swap pattern for switching models without restarting the agent.

Chapter 5: Ollama and Local Sovereignty

If you have never run a large language model locally, the first time you do it is a small shock. Not because it is difficult. Because it is easy. You install Ollama, pull a model, and within twenty minutes you have a capable LLM answering

questions without a network request leaving your machine. The shock is that it works this well on consumer hardware.

This chapter is about that setup in production: the specific configuration on `inference-host-1`, the models Bob uses, and the patterns that make local inference operationally reliable.

Why Local Inference Wins for Regulated Data

The argument for cloud LLMs is convenience and capability. The largest models available through APIs are larger than anything you can reasonably run locally on a single GPU. If you need GPT-4-class capability for a general-purpose assistant, the cloud is still ahead on raw benchmark scores.

For SQL Server diagnostic analysis, that argument does not apply. The task is constrained. You are not asking the model to write poetry or translate Mandarin legal documents. You are giving it a structured diagnostic payload, wait stats, query plans, DMV snapshots, and asking it to reason within a well-defined domain. For that task, a well-tuned 14B to 32B parameter model running locally is competitive with much larger cloud models, and in some cases beats them because the local model can be prompted with more context without worrying about token costs.

At Dogsbarking, where the work from May 2024 through March 2026 involved managing deduplication pipelines for enterprise Salesforce clients, the data moving through the system was contact records, transaction history, and account data from companies operating under GDPR. Using GPT-4 or Claude 3 to assist with entity resolution code was fine, that was developer tooling, and the code itself was not customer data. But the actual deduplication work happened on records. If you wanted an LLM to help analyze why two records were not matching correctly, you had to show it the records. Showing it the records meant sending personally identifiable information to an external API. Under GDPR Article 28, that triggers a Data Processing Agreement with the AI vendor, a review of their subprocessor list, documentation in your Article 30 records of processing, and possibly notification to the relevant supervisory authority depending on the data category. For a client in the EU, that is not a theoretical compliance exercise. It is a real legal obligation with real consequences if you skip it.

At Entergy via ComTec, from May 2020 through April 2024, the constraint was NERC CIP. North American Electric Reliability Corporation Critical Infrastructure Protection standards classify information about bulk electric system assets, configurations, and operational state as Protected Cyber Asset data. Sending a SQL query plan that references a table named `GridOperationsHist` or `TransmissionLineMonitor` to an external API is a CIP-002 and CIP-004 conversation at minimum. The standard does not list "LLM inference API

calls" as a prohibited channel because NERC CIP was not written with LLM APIs in mind. But the underlying principle, that protected asset information should not traverse uncontrolled external paths, is clear. During the audit period, 100% compliance meant no ambiguous interpretations. If there was a reasonable argument that something violated the standard, you did not do it.

The local inference setup eliminates both problems by eliminating the external transmission entirely.

Three concrete data-residency wins for organizations in regulated industries:

1. Audit trail completeness. When query plans and T-SQL text never leave your network, your audit trail is entirely within your control. There is no third-party data processor to include in your GDPR data processing records, no cloud provider's retention policy to worry about, no subpoena that can reach data you never sent anywhere. Your CISO can sign off on "we process this data on our own hardware" with confidence.

2. Zero token-cost exposure. A cloud LLM charges per token. Bob runs monitoring cycles continuously, potentially dozens of times per hour across multiple SQL Server instances. At cloud API pricing, a high-frequency monitoring use case becomes expensive quickly. The marginal cost of one more monitoring cycle on `inference-host-1` is negligible after the hardware is purchased.

3. Network isolation for sensitive schemas. Database schema names, stored procedure names, and column names appear in query plans and diagnostic output. Even when data values are excluded from the diagnostic payload, schema information often has sensitivity. In financial services, healthcare, or any environment where schema design is considered proprietary, sending that information to an external API is a risk that requires a vendor agreement, privacy review, and ongoing monitoring. With local inference, the schema stays local.

It is worth being honest about where cloud LLMs still belong in this picture. At Dogsbarking, I used GitHub Copilot, GPT-4, Claude 3, and Gemini daily for developer work: writing Python, generating T-SQL patterns, reviewing code logic, drafting documentation. Those tools are excellent for that purpose. The code they help produce is not sensitive. The prompts are not customer data. The seam between "AI as developer tool" and "AI in the production data path" is important.

The distinction is about what data you are putting in the model's context. Code patterns and technical questions are fine for external APIs. Production query plans with table names that reveal regulated schemas are not. Bob sits in the production data path. That is where the line is.

The `inference-host-1` Configuration

The primary inference host runs Ubuntu 25.10 with Ollama installed as a systemd service. Here is the unit file in production:

```
# /etc/systemd/system/ollama.service
[Unit]
Description=Ollama LLM Server
After=network-online.target
Wants=network-online.target

[Service]
Type=exec
User=ollama
Group=ollama
ExecStart=/usr/local/bin/ollama serve
Restart=always
RestartSec=3
Environment=OLLAMA_HOST=0.0.0.0:11434
Environment=OLLAMA_ORIGINS=*
Environment=OLLAMA_MODELS=/var/lib/ollama/models
Environment=OLLAMA_NUM_PARALLEL=2
Environment=OLLAMA_MAX_LOADED_MODELS=2

[Install]
WantedBy=multi-user.target
# source: bob@c549c88
```

The `.env` file that Bob's agent reads to locate the inference host:

```
// ollama_config.json
{
  "url": "http://10.0.0.70:11434",
  "model": "gemma4:26b",
  "timeout": 900,
  "num_ctx": 16384,
  "temperature": 0.7,
  "keep_alive": "10m"
}
// source: bob@c549c88
```

The failover chain lives in `llm_config.json`:

```
// llm_config.json (failover chain)
{
  "failover_chain": [
    {"provider": "ollama", "priority": 0, "url": "http://10.0.0.70:11434", "model": "gemma4
    {"provider": "ollama", "priority": 1, "url": "http://10.0.0.71:11434", "model": "qwen2.5
```

```
  ],
  "default_cooldown_seconds": 60,
  "max_cooldown_seconds": 900
}
// source: bob@c549c88
```

The one-line health check for `inference-host-1`:

```
curl -sf http://10.0.0.70:11434/api/tags | python3 -c "import sys,json; d=json.load(sys.stdi
# source: bob@c549c88
```

This returns something like `OK 8 models` when the service is healthy. The CI pipeline runs this check before any test that requires inference.

Model Discovery with `/api/tags`

Ollama's `/api/tags` endpoint returns a JSON list of every model that has been pulled to the host. Bob queries this at startup to verify the models it needs are present and to emit a warning if the preferred model is missing and a fallback is available.

```python
import httpx
import os

def discover_models(host: str) -> dict[str, dict]:
    """Return a dict of model_name -> model_metadata from the Ollama host."""
    response = httpx.get(f"{host}/api/tags", timeout=10)
    response.raise_for_status()
    models = {}
    for m in response.json().get("models", []):
        models[m["name"]] = {
            "size": m.get("size", 0),
            "modified_at": m.get("modified_at", ""),
            "details": m.get("details", {}),
        }
    return models

# source: bob@c549c88
```

At startup, Bob calls `discover_models` on the primary host. If the primary is unreachable, the `_ollama_call()` function automatically walks the `llm_config.json` failover chain, trying each server in priority order. Each server has its own default model configured separately: `gemma4:26b` on `.70`, `qwen2.5-14b` on `.71`. The two hosts carry different model sets. The active server and model are tracked globally so the dashboard can show which host is currently serving inference.

Hot-Swapping Models Without Restarting the Agent

A key operational requirement for production monitoring is the ability to change which model is being used without stopping the agent. This matters during A/B testing of model responses, when a better model is pulled to the host, or when a model is generating poor-quality diagnostics and needs to be replaced with a fallback.

Bob reads its active model from `ollama_config.json`. Updating that file and using the `/api/settings/ollama` REST endpoint, or the web dashboard's Models tab, is enough to change the active model. The change takes effect on the next LLM call.

The hot-swap sequence:

```
# 1. Pull the new model (can run while agent is active)
curl -X POST http://10.0.0.70:11434/api/pull \
  -d '{"name": "gemma4:27b"}'

# 2. Verify it appears in the model list
curl -sf http://10.0.0.70:11434/api/tags | \
  python3 -c "import sys,json; [print(m['name']) for m in json.load(sys.stdin)['models']]"

# 3. Update Bob's model via the REST API
curl -X POST http://localhost:8000/api/settings/ollama \
  -H "Content-Type: application/json" \
  -d '{"model": "gemma4:27b", "url": "http://10.0.0.70:11434"}'

# source: bob@c549c88
```

The model pull in step 1 happens in the background. Ollama supports concurrent pulls while inference is running. The old model remains loaded in memory until the pull completes and you send the reload signal. There is no gap in monitoring coverage during the swap.

This pattern also works for downgrading. If `gemma4:26b` is producing inconsistent diagnostic output for your specific workload and an alternate model performs better, the swap takes about two minutes.

There is a failure mode in the hot-swap process worth understanding. If you issue the settings API call before the model pull in step 1 has fully completed, Bob may attempt to call an incomplete model file. The failure looks like a context window error rather than a file corruption problem. The fix is to verify the model appears in `/api/tags` before updating Bob's configuration. Step 2 in the sequence handles this. If you skip it to save time, that is the failure you get.

What Was Tried and Rejected

`gemma4:26b` did not arrive as the default on day one. The model selection went through several iterations, and the rejections tell you as much as the final choice.

Earlier builds used `qwen2.5` variants during the architecture's testing phase. Qwen2.5 models handle structured output cleanly and are fast on the RTX 3090. The series remains in the failover chain: `qwen2.5-14b` runs on `inference-host-2` and is the automatic failover model when the primary is unreachable. The Qwen3-Coder variant is used specifically for Bob's evolution growth goals, the ones where Bob is actually writing Python code to extend his own capabilities, because code-specialized models produce better results for that narrow task.

Smaller models in the 7B–8B range were tested on `inference-host-2` at the beginning, when the RTX 3090 was not yet in the picture. They are fast on the RTX 3060 and produce coherent structured output. The smaller models are adequate for simple triage, the kind where you have one dominant signal and a clear recommendation falls out. They lose resolution on multi-factor problems where the correct recommendation requires holding several concurrent findings in working context and reasoning about their interaction. Deadlock analysis is the clearest example: a deadlock involves at least two processes, two resources, and an ordering problem. Getting the victim chain right requires reasoning about all of those together, and the smaller models would frequently identify the victim correctly but get the contention source wrong.

The VRAM economics on the RTX 3090 work out cleanly. `gemma4:26b` fits comfortably in 24 GB of VRAM with headroom for a full diagnostic context window. The `OLLAMA_MAX_LOADED_MODELS=2` setting is available if you want to keep a secondary model resident, but in practice the primary model stays loaded continuously because the load time from disk is several minutes and the monitoring workload does not justify evicting it between cycles.

Model Selection Philosophy

The production setup on `inference-host-1` runs `gemma4:26b` as the default model for all agent work: chat, goal evaluation, monitoring diagnostics, and the Forge evolution pipeline. The reasoning: the model fits comfortably in 24 GB of VRAM, performs well on structured reasoning tasks, handles SQL constructs reliably, and produces output that is consistent enough to parse programmatically. Bob's own evolution pipeline has made 34 verified self-modifications to its `reason()` function while running against this model, so the model and the agent have effectively co-evolved.

The failover chain configured in `llm_config.json`:

1. `gemma4:26b` on `10.0.0.70:11434` (primary)
2. `qwen2.5-14b` on `10.0.0.71:11434` (automatic failover)

On `inference-host-2` at `.71`, the RTX 3060 has 12 GB VRAM. The practical ceiling there is `qwen2.5-14b` at Q4. It is not as capable as the 26B model for complex multi-signal reasoning, but it is adequate for most monitoring tasks and sufficient for alert triage.

The Sovereignty Argument

When Bob's inference runs on hardware you own, you know exactly what the system is doing. You can inspect every inference request. You can log every prompt and every response. You can audit the model weights to verify they are what you downloaded. You can run in an air-gapped environment if required. You can guarantee to a regulator, auditor, or skeptical CISO that the diagnostic data for your SQL Server instances has not been transmitted to any third party.

You cannot do those things with a cloud API.

The counterargument is that cloud models are more capable. That is true at the frontier. It is less true for domain-specific tasks at production query volumes. For the specific problem Bob solves, the local models are good enough, the hardware is affordable, and the sovereignty is complete.

The next chapter is where the pieces come together: the three-layer agent that takes sensor data from SQL Server, routes it through Ollama for analysis, and acts on what it finds.

Chapter 6: The Three-Layer Agent

Bob's architecture is a response to a specific failure mode. An AI that can diagnose but not act is a smart alarm. One that acts without isolating bad data is a liability. The three-layer design solves both problems by separating observation, reasoning, and action into distinct components with explicit handoffs between them.

This chapter documents the architecture, the code, and the operational patterns that make it work safely.

Architecture Overview

```
flowchart TB
    subgraph L1[Layer 1: Sensors]
```

```
        S1[SQL Monitor health probes]
        S2[Network telemetry]
        S3[Log streams]
    end
    subgraph L2[Layer 2: Reasoning]
        R1[Ollama LLM router]
        R2[MCP tool orchestrator]
        R3[Memory store]
    end
    subgraph L3[Layer 3: Actuators]
        A1[T-SQL remediation]
        A2[Service restarts]
        A3[Alerting and notifications]
    end
    L1 --> L2 --> L3
    L3 -->|feedback| L2
```

The three layers collect, reason, and act. The feedback loop from Layer 3 back to Layer 2 is what makes Bob an agent rather than a script.

Layer 1: Sensors

The sensor layer is a set of probes that query SQL Server DMVs and system state on a configurable schedule. Probes run as lightweight Python callables that return structured data. They do not call Ollama. They do not take action. They observe and report.

The core SQL Monitor probe queries the DMVs most relevant to performance variance detection:

```sql
-- Primary diagnostic snapshot probe
-- Captures wait stats, top queries, index usage, and memory pressure
-- in a single round trip to minimize monitoring overhead.

WITH WaitStats AS (
    SELECT
        wait_type,
        waiting_tasks_count,
        wait_time_ms,
        max_wait_time_ms,
        signal_wait_time_ms
    FROM sys.dm_os_wait_stats
    WHERE wait_type NOT IN (
        -- Filter out benign background waits
        'SLEEP_TASK', 'WAITFOR', 'BROKER_TO_FLUSH',
        'BROKER_TASK_STOP', 'CLR_AUTO_EVENT', 'CLR_MANUAL_EVENT',
```

```
        'DISPATCHER_QUEUE_SEMAPHORE', 'FT_IFTS_SCHEDULER_IDLE_WAIT',
        'HADR_WORK_QUEUE', 'ONDEMAND_TASK_QUEUE',
        'REQUEST_FOR_DEADLOCK_SEARCH', 'RESOURCE_QUEUE',
        'SERVER_IDLE_CHECK', 'SLEEP_DBSTARTUP',
        'SLEEP_DBRECOVER', 'SLEEP_MASTERDBREADY',
        'SLEEP_MASTERMDREADY', 'SLEEP_MASTERUPGRADED',
        'SLEEP_MSDBSTARTUP', 'SLEEP_TEMPDBSTARTUP',
        'SNI_HTTP_ACCEPT', 'SP_SERVER_DIAGNOSTICS_SLEEP',
        'SQLTRACE_BUFFER_FLUSH', 'SQLTRACE_INCREMENTAL_FLUSH_SLEEP',
        'WAIT_XTP_OFFLINE_CKPT_NEW_LOG', 'XE_DISPATCHER_WAIT',
        'XE_TIMER_EVENT'
    )
),
TopQueries AS (
    SELECT TOP 10
        qs.total_elapsed_time / qs.execution_count AS avg_elapsed_us,
        qs.total_worker_time / qs.execution_count AS avg_cpu_us,
        qs.execution_count,
        qs.total_logical_reads / qs.execution_count AS avg_logical_reads,
        SUBSTRING(qt.text, (qs.statement_start_offset/2)+1,
            ((CASE qs.statement_end_offset
                WHEN -1 THEN DATALENGTH(qt.text)
                ELSE qs.statement_end_offset
             END - qs.statement_start_offset)/2)+1) AS query_text,
        qp.query_plan
    FROM sys.dm_exec_query_stats qs
    CROSS APPLY sys.dm_exec_sql_text(qs.sql_handle) qt
    CROSS APPLY sys.dm_exec_query_plan(qs.plan_handle) qp
    ORDER BY qs.total_elapsed_time DESC
),
MemoryPressure AS (
    SELECT
        physical_memory_in_use_kb / 1024.0 AS memory_used_mb,
        page_fault_count,
        memory_utilization_percentage
    FROM sys.dm_os_process_memory
)
SELECT
    'wait_stats'    AS section, (SELECT * FROM WaitStats FOR JSON PATH) AS data
UNION ALL
SELECT
    'top_queries'   AS section, (SELECT * FROM TopQueries FOR JSON PATH) AS data
UNION ALL
SELECT
    'memory'        AS section, (SELECT * FROM MemoryPressure FOR JSON PATH) AS data;
-- source: bob@c549c88
```

This probe returns three JSON payloads in a single query execution. The probe runner deserializes them and stores the result in a `DiagnosticSnapshot` dataclass:

```
from dataclasses import dataclass, field
from datetime import datetime
import json

@dataclass
class DiagnosticSnapshot:
    captured_at: datetime
    wait_stats: list[dict] = field(default_factory=list)
    top_queries: list[dict] = field(default_factory=list)
    memory_pressure: dict = field(default_factory=dict)
    instance: str = ""

    def to_prompt_context(self) -> str:
        """Serialize snapshot to a compact context string for LLM consumption."""
        return json.dumps({
            "captured_at": self.captured_at.isoformat(),
            "instance": self.instance,
            "wait_stats": self.wait_stats[:15],  # cap for context window
            "top_queries": self.top_queries[:5],
            "memory_pressure": self.memory_pressure,
        }, indent=2, default=str)

# source: bob@c549c88
```

Probes run on a 60-second cycle for the primary metrics. The deadlock graph probe runs on a 5-minute cycle since deadlock events are captured in the system health session and do not need frequent polling.

The `to_prompt_context()` method on `DiagnosticSnapshot` is where the first editorial decision happens. The raw wait stats table from `sys.dm_os_wait_stats` can return hundreds of rows, depending on how long the instance has been running. Sending all of them to the LLM is wasteful and counterproductive: it fills context window with noise. The method caps wait stats at fifteen rows, selected by the probe's `WHERE` filter which already excludes background system waits, and caps top queries at five. The compression is diagnostic-lossless. The wait types you care about are going to be near the top by `wait_time_ms`. If you have fifteen relevant wait types simultaneously, you have bigger problems than context window management.

The network telemetry probe is simpler but worth describing because it answers a class of question the SQL probe cannot. SQL Server wait stats will tell you there is an `ASYNC_NETWORK_IO` problem. They will not tell you whether the problem is in the application, in the switch, or in the virtualization layer. The network probe uses Bob's `network_scan` inline tool (which shells out to nmap)

to check ping latency and host reachability from the agent host to `db-host` at `10.0.0.14`. If SQL Server shows elevated network waits and the network probe shows normal latency to `.14`, the problem is probably in the application. If both degrade together, the problem is below SQL Server and the model can say so.

Log stream data currently comes from the SQL Server error log via `sys.dm_os_ring_buffers` and from the Windows Application Event Log via the agent host's Windows Management Instrumentation interface. In practice, the log stream produces the most value during and after a deadlock event: the deadlock XML is extracted from the system health session, reformatted into a compact representation, and appended to the diagnostic context when present. Deadlock analysis is one of the cases where the 32B model earns its VRAM allocation. Deadlock graphs have enough structural complexity that smaller models frequently misread the victim chain or recommend index changes that address a symptom rather than the contention source.

Layer 2: Reasoning

The reasoning layer takes the structured output from Layer 1, builds a prompt, sends it to Ollama, and decides what to do with the response. This is the most complex part of the system.

The Diagnostic Prompt

The system prompt is the contract between the agent and the model. It defines what the model is supposed to do, what data it will receive, and what format the response should take.

```
DIAGNOSTIC_SYSTEM_PROMPT = """You are a SQL Server diagnostic agent. Your job is to analyze
diagnostic snapshots from a production SQL Server instance and identify performance problems
that require attention.

You will receive a JSON diagnostic snapshot containing:
- wait_stats: current wait statistics from sys.dm_os_wait_stats
- top_queries: top 10 queries by elapsed time with query text and plans
- memory_pressure: process memory utilization

Your response MUST be valid JSON with this exact structure:
{
  "severity": "NONE|LOW|MEDIUM|HIGH|CRITICAL",
  "primary_finding": "one sentence describing the most important finding",
  "findings": [
    {
      "category": "WAIT_STATS|QUERY_PLAN|MEMORY|INDEX|LOCK",
      "description": "specific finding",
```

```
      "evidence": "which metric or value led to this finding",
      "confidence": "LOW|MEDIUM|HIGH"
    }
  ],
  "recommended_action": "NONE|MONITOR|INVESTIGATE|APPLY_FIX|ESCALATE",
  "fix_candidate": null
}

If recommended_action is APPLY_FIX, fix_candidate must contain a T-SQL statement
that is safe to run. All other fields remain required even when severity is NONE.

Do not fabricate DMV columns or table names. If you are uncertain about a finding,
set confidence to LOW. Do not recommend APPLY_FIX unless you are HIGH confidence."""

# source: bob@c549c88
```

The structured JSON response is mandatory. An LLM that produces free-form text here cannot be integrated with the actuator layer. The system prompt enforces the contract.

There is a constraint in the system prompt worth explaining because it was learned the hard way: "Do not fabricate DMV columns or table names." Early versions of the system prompt did not include that instruction, and some models, particularly smaller ones, would produce findings that cited real-sounding but nonexistent DMV columns. `sys.dm_exec_query_stats.total_spill_reads` does not exist, but a 7B model under compression will invent it confidently. The instruction does not fully solve the problem, but it reduces hallucination frequency measurably because it makes the prohibition explicit. When a finding references a nonexistent column, the JSON parse succeeds but the evidence string fails a validation check that runs against a whitelist of known DMV column names. The finding is downgraded to `LOW` confidence automatically.

Memory segmentation in the reasoning layer follows two patterns. Short-term memory is per-incident: when severity crosses `MEDIUM`, all findings from that incident are grouped by `incident_id` in the `FindingHistory` table and included in subsequent prompts until the incident is marked resolved. The model can see the progression within an incident. Long-term memory is per-host: the last 24 hours of findings for a given SQL Server instance are summarized into a rolling context string. The summary is not the raw JSON from each cycle; it is a prose paragraph generated by the model itself at the end of each hour, stored in `dbo.HostContextSummary`. This gives the model something it can reason with efficiently instead of having to re-parse 1,440 JSON objects to understand "what has this instance been doing today."

The Reasoning Router

Bob uses different models for different reasoning tasks. The router selects the model based on the complexity of the diagnostic payload:

```python
def route_to_model(snapshot: DiagnosticSnapshot, available_models: dict) -> str:
    """
    Select inference model based on snapshot complexity.
    Complex snapshots (many wait types, large query plans) go to the larger model.
    Simple snapshots (clean wait stats, small payload) can use a lighter model.
    """
    payload_chars = len(snapshot.to_prompt_context())

    has_query_plans = any(
        q.get("query_plan") for q in snapshot.top_queries
    )
    complex_wait_types = sum(
        1 for w in snapshot.wait_stats
        if w.get("wait_time_ms", 0) > 10_000
    )

    # Primary model for all work; qwen2.5-14b is the automatic failover
    # on inference-host-2 at .71 when the primary is unreachable
    preferred = ["gemma4:26b", "qwen2.5-14b"]

    if has_query_plans and payload_chars > 8000:
        # Log that this is a complex payload requiring full model capacity
        pass  # routing logic; complex flag used for telemetry

    for model in preferred:
        if model in available_models:
            return model

    raise RuntimeError("No suitable model available for inference")

# source: bob@c549c88
```

The routing logic has a practical origin. The first months of running Bob used a single model for everything: a mid-size model on `inference-host-2` at `.71`. It worked. Then `inference-host-1` came online with the RTX 3090 and `gemma4:26b` became available. Testing on the same diagnostic payloads showed the larger model produced noticeably better findings when query plans were in the payload. The plan XML adds tokens but also adds structure that the larger model uses to better advantage. The smaller model on identical inputs would sometimes identify the right problem but recommend the wrong fix. The larger model was more accurate about both.

The routing threshold at `payload_chars > 8000` was calibrated empirically. Below that size, the models produce essentially identical findings on clean workloads. Above that threshold, the divergence increases. The threshold is not sacred: it is a configuration parameter that should be adjusted if your workload has consistently large query plans. In production with `gemma4:26b` as the sole model on the primary host, the router effectively acts as a complexity gate that decides whether to call the primary or fall through to the failover.

Layer 3: Actuators and the Rollback-First Pattern

The actuator layer is where Bob takes action. It is also where things can go wrong in ways that matter. A bad recommendation is annoying; bad T-SQL on a production database is a data incident.

The design principle here is: **snapshot before you touch anything.**

Every potential fix that reaches the actuator layer follows this sequence:

1. Take a Proxmox snapshot of the SQL Server VM
2. Log the snapshot ID, timestamp, and the fix candidate to `BookOfBob.dbo.RemediationLog`
3. Apply the fix in a transaction with an explicit rollback window
4. Verify the expected metric improved
5. Mark the remediation as successful or roll back and alert

The snapshot-then-apply pattern in code:

```
import httpx
from dataclasses import dataclass
from datetime import datetime

@dataclass
class RemediationCandidate:
    fix_sql: str
    finding_description: str
    confidence: str  # LOW | MEDIUM | HIGH
    snapshot_id: str = ""

def execute_remediation(
    candidate: RemediationCandidate,
    proxmox_host: str,
    vm_id: int,
    sql_conn,
    db_conn,
) -> bool:
    """
    Execute a remediation candidate against the SQL Server instance.
    Always takes a Proxmox VM snapshot before applying any fix.
```

```
    Returns True if remediation succeeded, False if rolled back.
    """
    if candidate.confidence != "HIGH":
        raise ValueError(
            f"Will not auto-apply a fix with confidence={candidate.confidence}. "
            "Only HIGH confidence fixes may be auto-applied. Escalating instead."
        )

    # Step 1: Take Proxmox snapshot
    snap_name = f"bob-pre-fix-{datetime.utcnow().strftime('%Y%m%d-%H%M%S')}"
    response = httpx.post(
        f"{proxmox_host}/api2/json/nodes/pve/qemu/{vm_id}/snapshot",
        json={"snapname": snap_name, "description": candidate.finding_description},
        verify=False,
        timeout=60,
    )
    response.raise_for_status()
    candidate.snapshot_id = snap_name

    # Step 2: Log intent before touching anything
    db_conn.execute(
        """
        INSERT INTO dbo.RemediationLog
            (snapshot_id, fix_sql, finding, confidence, applied_at, status)
        VALUES (?, ?, ?, ?, SYSUTCDATETIME(), 'APPLYING')
        """,
        candidate.snapshot_id,
        candidate.fix_sql,
        candidate.finding_description,
        candidate.confidence,
    )
    db_conn.commit()

    # Step 3: Apply in a transaction with a rollback window
    try:
        sql_conn.execute("BEGIN TRANSACTION")
        sql_conn.execute(candidate.fix_sql)
        sql_conn.execute("COMMIT TRANSACTION")
        db_conn.execute(
            "UPDATE dbo.RemediationLog SET status='APPLIED' WHERE snapshot_id=?",
            candidate.snapshot_id,
        )
        db_conn.commit()
        return True

    except Exception as exc:
```

```
        sql_conn.execute("ROLLBACK TRANSACTION")
        db_conn.execute(
            "UPDATE dbo.RemediationLog SET status='ROLLED_BACK', error=? WHERE snapshot_id=?
            str(exc),
            candidate.snapshot_id,
        )
        db_conn.commit()
        return False

# source: bob@c549c88
```

The snapshot-then-apply pattern came from a production incident, not from upfront design. The original actuator implementation skipped the snapshot step for what seemed like a low-risk change: an `UPDATE STATISTICS` call on a table that the model had identified as having stale statistics. The statement executed. Statistics updated. Query performance got worse for about forty minutes until the query plan cache flushed and new plans were compiled. Not a disaster. Nothing corrupted. But forty minutes of degraded performance on a production database because Bob made a confident recommendation that turned out to be partially wrong.

That incident produced two changes. First, the snapshot requirement became non-negotiable for every actuator call, not just the "riskier" ones. The definition of risky is hard to draw in advance. The cost of a Proxmox snapshot is a few seconds. The cost of not having one when you need it is a manual restore from backup. Second, the confidence floor for `UPDATE STATISTICS` was raised from `MEDIUM` to `HIGH`. Statistics updates are cheap and frequent, but "cheap and frequent" is how you get a pattern of confident-but-wrong auto-applications that erode trust.

Two things to note about the code.

First: the confidence guard at the top of the function. Bob will not auto-apply a fix with `LOW` or `MEDIUM` confidence. Those candidates are logged and routed to the alert channel where a human reviews them. Only `HIGH` confidence fixes, and only certain categories of fix, are eligible for automatic application. The initial deployment restricts auto-apply to index maintenance operations: `ALTER INDEX ... REBUILD` and `UPDATE STATISTICS`. Everything else, query rewrites, configuration changes, service restarts, requires human approval.

Second: the Proxmox snapshot is taken before the transaction opens, not inside it. If the transaction commits and something is still wrong, the snapshot exists to roll back to. The snapshot ID is in the remediation log. The rollback path is: open Proxmox at `10.0.0.2`, find the snapshot by name, restore it. That is a manual process today. Chapter 8 covers automating it.

The confidence threshold for auto-apply is currently `HIGH` only. That threshold was three. There was a period during development when `MEDIUM` confidence fixes were allowed to auto-apply if the fix category was index maintenance. The

failure rate during that period was about one in eight. One in eight sounds acceptable until the eighth one runs at 7 PM on a Thursday and blocks a batch job for ninety minutes. `HIGH` only has a failure rate closer to one in fifty over the time Bob has been in production, and most of those failures are post-apply verification failures, which means the fix applied without error but did not produce the expected metric improvement. Those are safe failures. The metric did not get worse. They just did not get better.

The read-only mode mentioned in the error handling section is worth describing concretely. When Bob enters read-only mode, the probe cycle continues. Findings are logged to `BookOfBob.dbo.FindingHistory` as normal. Severity assessment continues. What stops is the handoff to the actuator layer. No `sql.apply_fix` calls, no `proxmox.snapshot` calls, no `notify.send` calls via Asterisk. The agent is watching but not touching. The alert that fires is a webhook to the configured notification channel, not an Asterisk call, since read-only mode is often triggered by inference host failure, and we cannot trust that the notification toolchain is fully intact either.

Why Three Layers and Not Two

The original prototype had two distinct components: the probe code that gathered SQL Server data, and the LLM call that processed it. No separate actuator layer. The LLM output was read and acted on in the same function. That worked at 200 lines. It broke at 1,000 lines when the complexity of acting correctly required logic that was too entangled with the logic of deciding what to do.

A two-layer design, sensors feeding directly into a combined reasoning-and-acting component, makes it hard to test the reasoning separately from the acting. You cannot verify that the model's diagnostic output is correct without also verifying the actuator response. You cannot add a new sensor without touching the action logic. You cannot swap the inference backend without touching the code that applies fixes. All of those things are changes you will need to make repeatedly as the system evolves.

Four layers was considered briefly: sensors, an evidence-processing layer that normalizes and filters data before it reaches the model, reasoning, and acting. The evidence-processing layer is real work and real value, but it is best implemented as a method on `DiagnosticSnapshot` rather than as a separate architectural layer. `to_prompt_context()` is the evidence processor. It is in the sensor layer, not between the sensor and reasoning layers, because it is fundamentally about what the sensor produces, not about how the reasoning layer consumes it.

Each layer does one conceptual thing. Sensor code that starts making decisions is violating the principle. Reasoning code that starts taking actions is violating

the principle. Three layers with explicit handoffs keeps those violations from happening silently as the codebase grows.

Memory and State

A monitoring agent without memory is stateless by definition. Stateless is clean, but it loses important information: was this wait stat elevated yesterday? Has this query been degrading over a week? Is this the third time this month we have seen this specific deadlock pattern?

Bob's memory store is a set of tables in `BookOfBob`:

```
-- Stores diagnostic snapshots for trend analysis
CREATE TABLE dbo.SnapshotHistory (
    snapshot_id     INT             IDENTITY(1,1) PRIMARY KEY,
    captured_at     DATETIME2       NOT NULL,
    instance        NVARCHAR(200)   NOT NULL,
    payload_json    NVARCHAR(MAX)   NOT NULL,
    severity        VARCHAR(10)     NOT NULL,
    primary_finding NVARCHAR(500)   NULL
);

-- Stores LLM findings for correlation
CREATE TABLE dbo.FindingHistory (
    finding_id      INT             IDENTITY(1,1) PRIMARY KEY,
    snapshot_id     INT             NOT NULL REFERENCES dbo.SnapshotHistory(snapshot_id),
    category        VARCHAR(20)     NOT NULL,
    description     NVARCHAR(MAX)   NOT NULL,
    evidence        NVARCHAR(MAX)   NOT NULL,
    confidence      VARCHAR(10)     NOT NULL,
    created_at      DATETIME2       NOT NULL DEFAULT SYSUTCDATETIME()
);
-- source: bob@c549c88
```

When Bob builds a diagnostic prompt, it includes a rolling window of the last 24 hours of findings from `FindingHistory`. This gives the model context: "four hours ago you reported a PAGEIOLATCH_SH issue on this instance; wait stats are now normal; was the issue self-resolving?" The model reasons about trends, not just snapshots.

The memory store is intentionally simple: it is a SQL Server database, queryable with standard T-SQL, not a vector database or embedding store. For the current use case, SQL is sufficient and has the advantage of being easily queried by the DBA without installing anything new.

Error Handling and the Fallback Path

When any component of the three-layer system fails, Bob degrades gracefully rather than silently.

The failure modes and their handling:

Inference host unreachable: Bob retries three times with exponential backoff, then switches to the fallback host at `.71`. If both are unreachable, Bob enters read-only monitoring mode: probes continue, findings are logged, no actuator actions are taken. An alert fires via the Asterisk PBX to the configured extension.

Probe query fails: The snapshot is logged as `PARTIAL`. The reasoning layer receives a note that the probe failed, includes the error, and sets severity to `MEDIUM` regardless of what other data says, because partial data is less reliable than complete data.

Actuator transaction fails: As shown in the code above, the transaction rolls back, the failure is logged, and the agent escalates to human review. The Proxmox snapshot that was taken before the attempt remains available for manual rollback if needed.

Model returns malformed JSON: The reasoning layer catches JSON parse errors, logs the raw response for review, and does not propagate a finding to the actuator. A malformed response is treated the same as a `NONE` severity with a note that the model's response was unparseable.

These failure modes map directly to the failure domain table in Architecture §2.5. Every domain listed there has a corresponding code path in the agent.

There is one failure mode not in that table because it took several months to name: model drift. Model drift is when the inference model produces findings that are technically coherent but systematically miscalibrated for your specific workload. Drift is systematic miscalibration, not hallucination. The model reads real DMV columns through a lens that does not match your environment. An example: `CXPACKET` waits are expected and normal on a server with heavy parallel query usage. A model that has not been calibrated for that environment will flag them as a problem. The model is not wrong in the abstract; `CXPACKET` waits can indicate parallelism problems. But on a reporting server with twelve-core parallelism and a heavy overnight batch, they are background noise.

The solution is not more model tuning. The solution is the system prompt addendum: a per-instance section that documents known baseline patterns the model should ignore. For `db-host` at `.14`, the addendum includes: "CXPACKET waits are expected during the overnight batch window from midnight to 5 AM and should not trigger alerts below severity HIGH. The top-5 queries by elapsed time include a nightly maintenance job that has a consistently high elapsed time and is not a performance regression." That addendum is maintained by the DBA,

not generated by the model. It is the human's domain expertise, encoded as structured instruction, sitting in the system prompt.

The three-layer architecture is what separates Bob from a monitoring script. The clean separation between sensing, reasoning, and acting means you can change any layer independently. You can swap the inference model without changing the probe code. You can add a new probe without changing the actuator logic. You can extend the actuator layer with new capabilities without touching the reasoning layer.

That modularity is where Chapter 7 begins. The Model Context Protocol is the mechanism that makes the reasoning layer's capabilities extensible without the agent knowing in advance what tools it will have.

Chapter 7: MCP as the Universal Bridge

A language model is a text-in, text-out system. It cannot run a SQL query. It cannot take a Proxmox snapshot. It cannot send an alert to an Asterisk extension. It produces text describing what it would do, and then a human or a downstream system decides whether to actually do it.

The Model Context Protocol, MCP, is the layer that closes that gap. It defines a standard way for a language model to declare the tools it can call and to receive the results of those calls. The model says "call this tool with these arguments," and the MCP server executes the call. The result comes back as structured data the model can reason about.

This chapter documents how MCP is integrated in Bob and how to extend it with a new tool. The end-to-end walkthrough in the last section should be completable in thirty minutes by a developer who has not worked with Bob before.

What MCP Solves

Before MCP, the integration between an LLM and external systems looked like one of three things.

The first approach was prompt injection: include all the data the model might need in the prompt, let it analyze it, and parse the response to decide what to do. This works for simple cases. It breaks down when you need to take multiple steps (get data, analyze it, fetch more data based on the analysis, act), because each step requires a round trip through the human or the orchestrating code.

The second approach was custom tool calling: define a JSON schema for the tools the model can call, add that schema to the context, and parse `{"tool":`

`"...", "args": {...}}` blocks out of the model's response. This works and was what Bob used before MCP. The problem is that every agent needs to implement this parsing logic differently, and every model handles the tool-calling format differently.

What the custom approach looked like in practice: the system prompt included a section that described available tools in plain text, the model was instructed to respond with a specific JSON structure when it wanted to call a tool, and the agent code had a parsing function that tried to extract tool calls from the model's response. That parsing function was about 80 lines of regex and JSON parsing logic, and it broke in two or three specific ways depending on which model was responding. `qwen2.5` wrapped the tool call in markdown code fences. `llama3.1` sometimes put the tool call inside a prose paragraph. One model version added a trailing comma to the JSON. Each quirk required a fix in the parsing code, and each fix was version-specific, so upgrading a model meant re-testing the parsing layer.

The transition to MCP cost one weekend. The bespoke parsing logic went away. The protocol handles the serialization and deserialization on both sides. The models that support MCP natively use the same wire format. The models that do not support it natively get a system prompt that describes the MCP tool schema, which is at least standardized XML even if the model has to be taught to use it. Upgrading from one Ollama model to another after MCP: no parsing code changes, zero.

The third approach, MCP, standardizes the tool interface. The model talks to the MCP server over a standard protocol. The server handles authentication, argument validation, and execution. The model receives structured results. Any model that speaks MCP can use any MCP server's tools without custom integration code.

Bob's Tool Architecture

Bob has 234 tools in total: 24 inline tools baked directly into the agent and 210 additional tools loaded dynamically from 70+ feature modules at startup. The inline set covers the core operations the agent needs on every task: reading files, running whitelisted shell commands, querying memory, managing Docker containers, scanning the network, and calling any registered MCP server through the `mcp_call` tool.

The `mcp_call` tool is the gateway to 43 registered MCP servers covering infrastructure (Proxmox, Docker, systemd), databases (SQLite, PostgreSQL, MSSQL, MySQL), network management (nmap, DNS), notifications, and more. The agent's system prompt includes a dynamically generated summary of every MCP server's available tools, grouped by category, so the LLM knows what it can ask for before it starts reasoning.

For SQL Server specifically, the `mcp_call` tool reaches the MSSQL MCP server which exposes 19 tools covering full CRUD, schema inspection, and diagnostics including active blocking detection, top CPU queries, backup freshness, wait statistics, and ad-hoc parameterized queries. The SQL Server connection uses a dedicated service account with `VIEW SERVER STATE`, `VIEW DATABASE STATE`, and write access to `BookOfBob` only. Not `sysadmin`. Not `db_owner` on production databases. The principle of least privilege is not a compliance checkbox here; it is the practical limit on what the agent can do even in a worst-case scenario.

The authentication model is worth a specific note because the open finding in `NETWORK_SCAN.md` for anonymous LDAP binds on `nas` at `.20` is a reminder of what happens when authentication is not explicit. Anonymous binds on the LDAP service mean any host on the LAN can query directory information without credentials. The Bob toolchain does not repeat that mistake. MCP server access requires Ward's approval via the web dashboard before Bob can call any tools on that server. A request flows through the `mcp_request` endpoint, Ward approves or denies in the dashboard, and only approved servers are callable.

Every tool call is subject to the authority matrix. Read-only calls (listing containers, reading files, querying network state) execute at the `auto` tier with no notification. Mutating calls (restarting services, writing files outside Bob's workspace) require Ward's approval via a Kanban card in the web dashboard's Todo lane. The blast radius of every rule is documented in `authority_matrix.json`. If Bob encounters an action that does not match any rule, the default tier is `ask`. It queues the action for human review rather than proceeding.

MCP Architecture in Bob

```
flowchart LR
    llm[Ollama LLM] -- "THOUGHT/ACTION" --> react[ReAct Loop]
    react -- "mcp_call" --> dispatch[MCP Dispatch]
    dispatch -- "approval check" --> auth[mcp_state.json]
    dispatch -- "route" --> tools{43 MCP Servers}
    tools --> mssql[MSSQL MCP server]
    tools --> prox[Proxmox MCP server]
    tools --> net[Network tools]
    tools --> notify[Notification tools]
    mssql --> db-host[db-host .14:50003]
    prox --> proxmox[Proxmox .2]
    net --> dns[dns-primary/dns-secondary .222/.221]
    notify --> hermes[Hermes webhook]
```

The MCP dispatch module is a native Python layer. No subprocess overhead. It communicates with MCP servers over stdio or network sockets as configured in

`mcp_servers.json`. MCP server approval state is tracked in `mcp_state.json`. Bob maintains a pool of persistent MCP server connections for low-latency tool calls. The catalog of all 43 servers and their available tools is rebuilt on demand via `/api/mcp/refresh-catalog`.

Adding a New Tool: End-to-End Walkthrough

This walkthrough adds a new dynamic tool, `sql_index_analysis`, that returns fragmentation statistics for all indexes in a given database. Bob loads dynamic tools at startup from feature modules: Python packages that expose a `get_tools()` function. Adding a capability is three files and a registration entry. The whole process takes about twenty-five minutes the first time.

Prerequisites: Python 3.13+, access to the Bob agent host, read access to the repository at `http://10.0.0.80:3000/hyp3rsoft/bob`.

Step 1: Create the Feature Module (5 minutes)

Create a directory `sql_index_analysis/` in the Bob project root and add `tools.py`:

```
# sql_index_analysis/tools.py
import pymssql

def get_tools() -> list[dict]:
    return [
        {
            "name": "sql_index_analysis",
            "fn": _sql_index_analysis,
            "description": (
                "Query index fragmentation statistics for a SQL Server database. "
                "Returns indexes with fragmentation above the threshold. "
                "Use this to identify indexes that need REBUILD or REORGANIZE."
            ),
        }
    ]

def _sql_index_analysis(database_name: str, min_fragmentation_pct: float = 10.0) -> str:
    """
    Returns index fragmentation data for a database.
    Safe: read-only, no modification of database state.
    """
    import json, os
    host = os.environ.get("MSSQL_HOST", "10.0.0.14")
    port = int(os.environ.get("MSSQL_PORT", 50003))
    user = os.environ.get("MSSQL_USER", "bob_agent")
```

```
    password = os.environ.get("MSSQL_PASSWORD", "")

    try:
        conn = pymssql.connect(host, user, password, database_name, port=port)
        cursor = conn.cursor(as_dict=True)
        cursor.execute(
            """
            SELECT
                OBJECT_NAME(ips.object_id)          AS table_name,
                i.name                               AS index_name,
                ips.avg_fragmentation_in_percent,
                ips.page_count,
                ips.index_type_desc
            FROM sys.dm_db_index_physical_stats(
                DB_ID(%(db)s), NULL, NULL, NULL, 'LIMITED'
            ) ips
            JOIN sys.indexes i
                ON ips.object_id = i.object_id
                AND ips.index_id = i.index_id
            WHERE ips.avg_fragmentation_in_percent > %(pct)s
              AND ips.page_count > 100
            ORDER BY ips.avg_fragmentation_in_percent DESC
            """,
            {"db": database_name, "pct": min_fragmentation_pct}
        )
        rows = cursor.fetchall()
        conn.close()
        return json.dumps({
            "success": True,
            "database": database_name,
            "fragmented_indexes": rows,
            "count": len(rows),
        }, default=str)
    except Exception as exc:
        return json.dumps({"success": False, "error": str(exc)})

# source: bob@c549c88
```

Add `__init__.py` to make it a package:

```
# sql_index_analysis/__init__.py
from .tools import get_tools
# source: bob@c549c88
```

Step 2: Register the Module (2 minutes)

Open `self_agent.py` and find the `_register_module_tools()` function. Add the module name to the `modules_with_tools` list:

```
modules_with_tools = [
    # ... existing modules ...
    "sql_index_analysis",   # add this line
]
# source: bob@c549c88
```

Step 3: Add a Test (10 minutes)

Create `tests/tools/test_sql_index_analysis.py`:

```
from unittest.mock import patch
import json
from sql_index_analysis.tools import _sql_index_analysis

def test_returns_fragmented_indexes():
    mock_rows = [{
        "table_name": "Orders",
        "index_name": "IX_Orders_CustomerId",
        "avg_fragmentation_in_percent": 45.2,
        "page_count": 1024,
        "index_type_desc": "NONCLUSTERED INDEX",
    }]

    with patch("sql_index_analysis.tools.pymssql") as mock_pymssql:
        mock_conn = mock_pymssql.connect.return_value
        mock_conn.cursor.return_value.fetchall.return_value = mock_rows
        result = json.loads(_sql_index_analysis("AdventureWorks", 30.0))

    assert result["success"] is True
    assert result["count"] == 1
    assert result["fragmented_indexes"][0]["avg_fragmentation_in_percent"] == 45.2

def test_handles_connection_error():
    import pymssql
    with patch("sql_index_analysis.tools.pymssql") as mock_pymssql:
        mock_pymssql.connect.side_effect = pymssql.OperationalError("connection failed")
        result = json.loads(_sql_index_analysis("AdventureWorks"))

    assert result["success"] is False
    assert "connection failed" in result["error"]

# source: bob@c549c88
```

Run the tests:

```
cd /home/ward/Projects/08-MY-AGENTS/Bob
.venv/bin/python -m pytest tests/tools/test_sql_index_analysis.py -v
# source: bob@c549c88
```

Step 4: Restart the Web Server (2 minutes)

```
# The web server loads all feature modules at startup
pkill -f "uvicorn web_server"
.venv/bin/python -m uvicorn web_server:app --host 0.0.0.0 --port 8000 &

# Verify the tool registered successfully
curl -s http://localhost:8000/api/stats | python3 -c \
  "import sys,json; d=json.load(sys.stdin); print('tools:', d.get('tool_count','check /api/t
# source: bob@c549c88
```

Step 5: Test with a Live Chat Call (5 minutes)

```
curl -s http://localhost:8000/api/chat \
  -H "Content-Type: application/json" \
  -d '{
    "message": "Check index fragmentation in the BookOfBob database on db-host and tell me w
    "model": "gemma4:26b"
  }' | python3 -m json.tool
# source: bob@c549c88
```

Bob's ReAct loop will reason about the request, call `sql_index_analysis` with the appropriate arguments, observe the fragmentation data, and synthesize a diagnostic response. You should see the THOUGHT / ACTION / OBSERVATION trace in the response stream.

The walkthrough above describes the happy path. There are several failure modes a first-time implementer will encounter.

The most common one is the tool not appearing after restart. If the tool registered without errors, check whether the module name was added to the correct `modules_with_tools` list and that the `__init__.py` exports `get_tools`. Run `python -c "from sql_index_analysis import get_tools; print(get_tools())"` to verify the import chain before restarting the server.

The second common failure mode is a database connection error when the tool actually runs. The unit test mocks the connection entirely, so it passes regardless of credential validity. The live call does not mock the connection. Check that `MSSQL_HOST`, `MSSQL_PORT`, `MSSQL_USER`, and `MSSQL_PASSWORD` are set correctly in the environment before assuming the tool code is wrong.

The third failure mode is the agent describing index fragmentation analysis in

its response without calling the tool. This means the LLM recognized the task but did not select `sql_index_analysis` from the tool list. It often means the description string in `get_tools()` does not match the language in the request well enough. Revise the description to include the key terms the user is likely to use ("fragmentation," "REBUILD," "defragment") and test again.

Tool Design Guidelines

A few principles from building the existing tool set:

1. Keep tools atomic. One tool does one thing. `sql.run_diagnostic` runs a query and returns results. It does not analyze the results. The model does the analysis. Mixing execution and analysis in a single tool makes the tool harder to test and harder to trust.

2. Fail loudly. A tool that returns `{"success": false, "error": "connection refused"}` is better than one that returns empty results or swallows exceptions. The model needs to know when a tool failed so it can reason about the failure, not silently treat missing data as a clean bill of health.

3. Log every call. Every tool call in Bob is logged. The `outcomes.json` file records every autonomous action with goal ID, action taken, result, status, and duration. The `llm_snoop.jsonl` file captures every LLM call across all processes (prompt, response, model, timing, and caller). The `notify_log.jsonl` captures every outbound notification. Every significant action has at least one durable record.

4. Respect the authority matrix. Every tool that takes a mutating action should check authority via `check_authority` before executing. Tools that might change system state outside Bob's workspace should land at `notify` or `ask` tier, not `auto`. The authority matrix in `authority_matrix.json` defines what Bob can do autonomously versus what requires Ward's explicit approval via the Kanban board.

5. Return structured data, not narrated summaries. A tool that returns `{"fragmentation_pct": 45.2, "index_name": "IX_Orders_CustomerId"}` gives the model something to reason with. A tool that returns `"The Orders table has a nonclustered index called IX_Orders_CustomerId that is 45% fragmented"` gives the model something to regurgitate. The difference matters at the reasoning layer: structured data enables the model to compare values, rank findings, and make logical decisions. Narrated summaries push the model toward pattern-matching on text, which is less reliable than arithmetic on numbers. Every tool in Bob returns the former.

The MCP architecture has no natural ceiling on what Bob can call. Any system reachable from the agent host, any SQL Server instance, any Proxmox VM, any service with an HTTP API, can become a tool. The constraint is discipline: every new tool is a new attack surface if the input validation is weak, and a new liability if the logging is missing.

Part 3 starts with the question that follows from all of this working: what happens when Bob is capable enough to take over more of the operational load, and how do you hand that over safely?

Chapter 8: Toward Self-Healing

The phrase "self-healing infrastructure" gets used loosely. It appears in vendor decks alongside phrases like "zero-touch operations" and "autonomous remediation," and like most marketing language, it describes something real and hides what building it actually costs.

A self-healing system detects a specific class of problems, diagnoses them correctly, applies a known-good fix without human intervention, verifies the fix worked, and rolls back safely if it did not. Everything short of that complete loop is automation, which is valuable but different.

Bob, as of this writing, executes parts of that loop. Not all of it. This chapter is honest about where the line sits today and what it would take to move it.

What Bob Does Today

Bob's current operational loop is this: probes run on a 60-second cycle, collecting wait stats, query performance data, and memory pressure from the monitored SQL Server instance. Every cycle, the Layer 2 reasoning engine builds a diagnostic prompt, sends it to Ollama on `inference-host-1` at `10.0.0.70`, and parses the structured JSON response. If severity is `NONE` or `LOW`, the finding is logged and the cycle ends. If severity is `MEDIUM`, an alert fires and a human reviews the finding. If severity is `HIGH` or `CRITICAL`, the agent proceeds toward the actuator layer.

The actuator layer has a gatekeeper: the confidence check. Bob will not auto-apply any fix the model rates below `HIGH` confidence. And even within high-confidence findings, the current auto-apply scope is narrow on purpose: only index maintenance operations, `ALTER INDEX ... REBUILD` and `UPDATE STATISTICS`, run automatically. Everything else, query rewrites, configuration changes, service restarts, requires a human to read the finding and give explicit approval through the management interface.

The Proxmox snapshot always comes first. This is enforced in the remediation tool code: the tool will refuse to execute if no snapshot call appears in the current

remediation log for this session. That check runs against the remediation log on `db-host` at `10.0.0.14:50003`. No snapshot record means no apply. Period.

This architecture means that today Bob reliably handles one class of problem autonomously: index fragmentation. High-fragmentation indexes rebuild overnight without anyone waking up to do it. Everything else is assisted: Bob identifies it, explains it, and waits.

The Failure Domain Map

Understanding where Bob sits on the self-healing spectrum requires understanding what can go wrong and what the system does when it does. The failure domains from Architecture §2.5 are the frame:

LLM host down. If `inference-host-1` at `.70` is unreachable, Bob's `_ollama_call()` function automatically walks the `llm_config.json` failover chain. No human configuration reload required. On the RTX 3060, the model ceiling drops. `gemma4:26b` cannot fit in 12 GB of VRAM. The fallback model on `.71` is `qwen2.5-14b`. Diagnostic quality degrades under heavy query plan loads, but monitoring continues. The heartbeat loop also has a circuit breaker: after three consecutive LLM failures, it stops attempting goal execution until the LLM recovers. This prevents cascading failures from a runaway retry loop. If both inference hosts are unreachable, Bob's heartbeat circuit breaker trips, monitoring probes continue, no autonomous actions are taken, and an alert fires via the notification system. The system fails safe, not silent.

MCP tool error. When a single tool call fails, the agent falls back to read-only mode for that capability. If the SQL diagnostic MCP call fails, the probe returns a partial snapshot. If the Proxmox snapshot call fails, the remediation gate blocks: no snapshot record means no apply. The agent escalates and waits. This is the right behavior. A monitoring system that applies fixes when it cannot snapshot first is more dangerous than a monitoring system that does nothing.

SQL fix misfires. The snapshot pre-apply requirement is the primary defense here. The secondary defense is the transaction boundary in `execute_remediation`: the fix runs inside `BEGIN TRANSACTION / COMMIT TRANSACTION`. If the SQL itself executes without a runtime error but produces a wrong result, the transaction commits, the snapshot remains available for manual rollback, and the post-apply verification step should catch the problem. Post-apply verification currently checks whether the targeted metric improved within five minutes. If the metric is the same or worse, Bob marks the remediation as `VERIFY_FAILED`, fires an alert, and logs the snapshot ID for manual rollback. The snapshot name format is `bob-pre-fix-YYYYMMDD-HHMMSS`, easily identifiable in the Proxmox interface at `10.0.0.2`.

Network partition. If the agent host cannot reach the SQL Server or the Proxmox API, probes fail with connection errors. The agent treats a connec-

tion failure like a probe failure: partial data, elevated default severity, alert to the operations channel. Network-level detection beyond simple reachability is provided by Bob's `network_scan` tool (nmap-based) and the multi-host SSH monitoring that probes `dns-primary` at `.222`, `dns-secondary` at `.221`, and other configured hosts every monitoring cycle. A partition that isolates the agent from SQL Server but leaves the agent reachable is logged. A partition that takes the agent itself offline is not detectable from within the agent, which is expected: monitoring cannot monitor itself. The notification system at P0 severity handles the escalation path for events the agent cannot self-report.

```
flowchart TD
    probe[Probe cycle starts]
    probe --> infer{Inference host reachable?}
    infer -- Yes --> reason[Layer 2 reasoning]
    infer -- No --> retry[Retry x3 with backoff]
    retry -- Primary still down --> fallback[Switch to inference-host-2 .71]
    retry -- Primary recovered --> reason
    fallback -- Fallback also down --> readonly[Read-only mode + alert]
    fallback -- Fallback ok --> reason
    reason --> severity{Severity?}
    severity -- NONE/LOW --> log[Log and continue]
    severity -- MEDIUM --> alert[Alert + human review]
    severity -- HIGH/CRITICAL --> gate{Confidence HIGH?}
    gate -- No --> alert
    gate -- Yes --> snap{Snapshot successful?}
    snap -- No --> alert
    snap -- Yes --> apply[Apply fix in transaction]
    apply --> verify{Metric improved?}
    verify -- Yes --> done[Log success]
    verify -- No --> escalate[Alert + log snapshot ID for manual rollback]
```

What Already Self-Heals: The Forge Pipeline

The gap between where Bob is today and a fully self-healing system is not a gap in the agent logic. Several pieces of that loop are already running in production, and they are worth understanding before describing what still requires human judgment.

The most significant self-healing mechanism is one most DBAs would not think to look for: Bob rewrites his own source code. The Forge evolution pipeline runs in a separate process (`run_evolution.py`) on a 30-cycle batch schedule managed by a systemd timer. Each cycle follows seven steps (DIAGNOSE, TARGET, GENERATE, VALIDATE, EVALUATE, APPLY, REFLECT) and uses approximately 16–18 LLM calls to challenge Bob's own reasoning, generate candidate improvements, and apply the best one if it passes all safety gates.

The APPLY step has 8 sequential safety layers before any patch reaches disk: non-empty check, existence verification, uniqueness check, syntax compilation, essential component verification, SHA-256 hash verification of 16 sacred functions that can never be modified, semantic invariant checks, and a timestamped backup creation. After the patch is written, two runtime checks run: 7 behavioral smoke tests in a subprocess with a 15-second timeout, and a scoring gauntlet of 3 real LLM-graded questions with a 120-second timeout. The patch must score an average of 7.0 or above, with no more than a 1.5-point drop from the recent 5-cycle average. Any failure at any layer triggers automatic rollback from the timestamped backup.

In production, the Forge pipeline has completed 1,027+ cycles and applied 34 verified modifications, all targeting the `reason()` function that drives Bob's diagnostic reasoning. Current diagnostic scores run 9.8–10.0 out of 10 on domain-relevant sysadmin questions. Those scores are what justify the autonomous monitoring posture: the model has been continuously validated against real questions, not just run in good faith.

The three maturity levels below apply to the SQL-specific remediation path: the path from finding an actionable SQL Server problem to applying a fix autonomously.

Level 1: Alert-only. This is where Bob started and where it ran for approximately three months. Every finding is logged. High-severity findings fire P0/P1 alerts via the notification system. Nothing applies automatically. The DBA reads the finding, decides what to do, and acts manually. This level produces the training data for Level 2: you accumulate a history of findings and their resolutions, which tells you how accurate the model's recommendations are before you give it any authority to act on them.

Level 2: Suggest-with-approval. The reasoning layer generates a specific fix candidate and presents it to the DBA through the Kanban board in the web dashboard. The DBA reviews the finding, the confidence score, the proposed SQL, and the snapshot status, then approves or denies the Kanban card. This level is where you build trust in the model's judgment for specific fix categories. Index maintenance recommendations were in this level for about two months before moving to Level 3. Every approval or denial is logged to `outcomes.json`, so you can look back and see that out of forty-seven index maintenance recommendations, forty-three were approved and applied successfully, three were rejected by the DBA as low-priority, and one was rejected because the model recommended rebuilding an index on a table that was actively receiving a bulk load.

Level 3: Apply-with-rollback. The current production level for index maintenance. The confidence gate, the snapshot pre-condition, and the post-apply verification step are what make this safe rather than reckless. The transition from Level 2 to Level 3 for a given fix category requires two things: a track record from Level 2 that shows a false-positive rate below roughly 5%, and an

operational window constraint that restricts auto-apply to hours when the risk of blocking is lowest.

The gating mechanism between levels is not a technical threshold. It is a decision by the DBA running the system, based on the evidence accumulated at the previous level. There is no algorithm that tells you when to promote a fix category from suggest to apply. You decide when you trust the track record you have built. That subjectivity is not a design flaw. The track record is the argument, not the algorithm.

This is how the roadmap is sequenced for the SQL remediation path:

Phase 1: Expand the auto-apply scope carefully. Index maintenance is conservative and low-risk. The next category is statistics updates, which are already on the auto-apply list. After that: automatically killing long-running blocking sessions that have been blocking for more than a configurable threshold (default: 15 minutes) and have no open transaction of their own. That action is reversible in practice (the session will reconnect if the application is designed correctly) but not reversible in the transactional sense. It requires confidence from both the model and an operational window check: do not auto-kill sessions during business hours unless severity is `CRITICAL`.

Phase 2: Cross-instance correlation. Today Bob monitors one SQL Server instance. The architecture supports multiple via `mssql_instances.json`, but the reasoning layer currently treats each instance independently. Cross-instance correlation means: if three instances all show the same wait type spiking at the same time, that is probably not a per-instance SQL problem, it is probably an infrastructure problem, a storage controller, a network segment, a shared service. Detecting and naming that correlation requires a reasoning step that runs across instances rather than within one.

Phase 3: SQL-specific evolution goals. Bob already has an autonomous evolution goal system (goals g013–g020 in `goals.json`) that has successfully built 8 new capabilities by writing Python code to disk and verifying they work. The next frontier is SQL Server-specific growth goals: an index fragmentation analyzer, a TempDB health monitor, a Query Store reader. These would follow the same pattern as the existing evolution goals: defined in `goals.json`, evaluated by the heartbeat using Qwen3-Coder, verified by file existence and import check. The SQL feature gap analysis in `docs/SQL_MONITOR_FEATURE_GAPS.md` already has 20 goal definitions ready to feed into this pipeline.

The Question of Trust

There is a principle in reliability engineering called the automation trust gradient. The idea is that you should extend autonomy to a system incrementally, proportional to the track record. You do not hand a new system the keys on

day one. You give it narrow authority, watch what it does with that authority, and expand the scope as confidence accumulates.

Bob started with zero auto-apply authority. Every finding was routed to a human. That phase lasted about three months. In three months of watching, the high-confidence index maintenance recommendations were correct roughly ninety percent of the time. The ten percent failures were mostly cases where the fragmentation was real but the index was on a table being actively loaded, so the rebuild caused brief blocking. Not disasters, but not clean either.

The solution was an operational window check: auto-apply index maintenance only between midnight and 5 AM. That check dropped the failure rate to near zero over the next two months. With that track record established, index maintenance moved to full autonomy.

That is the model for every subsequent expansion: narrow scope, operational window check, watch the failure rate, adjust. Automate what the track record supports, nothing more.

There is a category of fix that will never enter Level 3, not because the model cannot reason about it correctly, but because the judgment call it requires is inherently contextual in ways the model cannot fully see.

Schema changes are the clearest example. If Bob detects that a missing index would eliminate a 90% table scan on a critical query, the recommendation is correct. The risk is in the execution: adding an index on a large table during business hours blocks writes for the duration of the build. The model can know that index builds block writes. It cannot know whether a blocking event right now will affect a payment processing window, a report that a VP is waiting for, or a data load from an external partner that has contractual SLA implications. That context lives in the organization, not in the DMVs. A DBA who picks up the phone and asks before scheduling the index build has access to that context. An automated agent does not.

The same argument applies to query rewrites. A plan regression where the optimizer chose a hash join over a nested loop join because statistics are stale is a diagnosable, fixable problem. Bob can identify it. An `UPDATE STATISTICS` call might fix it. But if the query is in a stored procedure that belongs to a third-party application whose vendor has a support clause that prohibits schema modifications without their sign-off, the correct action is to email the vendor, not to rewrite the query. Bob cannot know that the stored procedure has a vendor lock. The DBA can.

The self-healing system that is worth building is one that handles the routine clearly and escalates the complex cleanly. The boundary between those two categories shifts over time as the model's track record grows and as the system's knowledge of the environment deepens. But it never disappears entirely. There will always be a class of judgment that belongs to a human with organizational context that no monitoring system can fully encode.

A DBA who has been in the field long enough has internalized this gradient through painful experience. Automation that outstrips its track record produces incidents. Incidents produce rollbacks, not just of code but of organizational trust in automation generally. Once an organization has been burned by a system that did too much too fast, getting approval for the next automation initiative is a year-long political project.

Bob is designed not to earn that kind of reputation. The aggressive gating, the snapshot requirement, the confidence floor, the operational windows: none of these are bureaucratic overhead. They are how you build a track record.

The next chapter is where the personal becomes the communal. The architecture documented in Part 2 is not meant to stay in one server room. The question Chapter 9 addresses is how to share it in a way that actually works: not as a GitHub repo that nobody runs, but as a living community practice.

What Bob does for one DBA's infrastructure scales. The bottleneck is the trust gradient, not the technology, multiplied across every organization that would need to build its own track record from scratch. Chapter 9 is about shortcutting that process without cutting corners.

Chapter 9: The AI-for-DBAs Community

Here is the chapter I cut.

It was a manifesto. It had sections with names like "The Coming Displacement" and "Reclaiming the DBA Role." It made arguments about professional survival. It spent a lot of time explaining why database administrators should care about AI rather than just showing them something worth caring about.

Manifestos age poorly. DBAs need something they can use on Monday morning.

What the Community Is

"AI for DBAs" is a working community of practice built around one concrete idea: the interpretation gap between raw diagnostic data and actionable decisions is closeable, and the people who should close it are the ones who understand both ends of that gap.

That means DBAs who understand SQL Server deeply enough to know which signals matter. Not AI researchers. Not platform engineers who have never read a deadlock graph. People who have been paged at 3 AM and solved the problem before business hours.

The community exists as a place to share what that combination produces: diagnostic patterns that work, system prompt templates refined against production

workloads, tool designs that do not break under the conditions real SQL Server environments create. Not theoretical architectures, but working code, against live systems, with honest notes on what failed and why.

The "AI for DBAs" identity listed under my volunteering work is not an honorary title. It is the active project. Bob is the technical artifact. The community is the mechanism by which Bob becomes more than one person's home lab experiment.

The Real Problem with DBA Knowledge

The skill distribution problem in the DBA profession is something Chapter 3 outlined: too many senior DBAs spread across too many instances, junior people who lack the pattern recognition that only comes from years of exposure, the middle layer that is genuinely rare and genuinely expensive. That distribution has not changed meaningfully in the decade I have been paying attention to it.

What has changed is the tooling available to close the gap.

When I started at Chickasaw Nation in 2007, the knowledge transfer mechanism was sitting next to the senior DBA and watching. That is still the best way to learn, and it is not scalable. By the time I was doing HIPAA compliance work at Lumeris in 2017, the situation was: one DBA (me), many databases, PHI on the line, and no one to consult for the edge cases except documentation and SQL Server forums. Not ideal.

At Entergy, through ComTec from 2020 to 2024, the monitoring toolchain was sophisticated but the interpretation still landed on humans. NERC CIP auditing requires 100% compliance, and achieving that required custom SSRS reports that a human read every week. The reports were good. The human-reading-them step was the bottleneck. I automated about 70% of the routine analysis away with PowerShell by the time I left, but the remaining 30%, the non-routine findings that required judgment, still required a DBA.

That 30% is what Bob addresses. And it is where the community work matters most, because the knowledge required to handle that 30% is distributed across thousands of people who have been doing this work for years. It is not in any single repository. It is in the collective memory of a profession.

What Sharing Actually Requires

DBAs share knowledge generously on forums. Ask a question about a wait stat on DBA Stack Exchange and you will get a good answer within hours. Ask for someone's complete monitoring agent with deployment instructions and you will get silence.

The gap between "willing to share a query" and "willing to share a system" is real and has multiple causes.

First: context collapse. A diagnostic query is self-contained. A monitoring system embeds assumptions about the environment: schema names, connection strings, operational windows, escalation contacts. To share it as something another DBA can actually use, you have to strip the context out and replace it with configuration points. That is real work.

Second: embarrassment. The code I wrote in the early Bob prototypes was functional and ugly. The kind of code you do not want anyone to read. Getting it to the point where you are comfortable having someone else run it requires a refactor pass that takes time.

Third: liability anxiety. If you share a system that applies fixes automatically and someone runs it against a production database they do not fully understand and something breaks, whose fault is that? The cultural answer in the DBA community has historically been: not worth the risk.

The community I am building handles these by changing what "sharing" means. The primary artifacts are not finished systems. They are patterns: diagnostic patterns that work in certain conditions, prompt templates refined for specific failure classes, tool designs that have been validated in production. Those are shareable without the full context problem. They require the person who uses them to do the integration work, which means they understand what they are running before they run it.

The Prompt Library

The most immediately valuable thing the community produces is a library of system prompt templates, each annotated with the failure class it was designed for, the model it was tested against, and the production conditions under which it worked.

Not vague templates. Precise ones. Here is the shape of a prompt template entry:

```
TEMPLATE: deadlock-cycle-diagnosis
FAILURE CLASS: Recurring deadlock on same two tables
TESTED AGAINST: qwen2.5:14b, qwen2.5:32b
PRODUCTION CONDITIONS: OLTP workload, 50-200 concurrent sessions, SQL Server 2019

SYSTEM PROMPT:
You are analyzing a SQL Server deadlock graph. The deadlock involves two or more
sessions blocking each other in a circular dependency. Your task is to identify:
1. Which two objects (tables, indexes) are at the center of the cycle
2. Whether the cycle pattern suggests a missing index, an ordering problem in
```

```
   application code, or a transaction isolation level mismatch
3. A specific T-SQL test query that would confirm your hypothesis

Output as JSON: {"root_cause_hypothesis": "...", "confidence": "LOW|MEDIUM|HIGH",
"confirmation_query": "...", "recommended_fix": "..."}

Do not fabricate table or index names. Use only what appears in the deadlock graph.

NOTES: This template performs poorly against deadlocks involving more than four
sessions. For high-cardinality deadlock graphs, use the deadlock-correlation template
which feeds the graph through a preprocessing step first.
# source: bob@c549c88
```

That level of specificity is what distinguishes a useful template from a general suggestion. The "NOTES" field is as important as the prompt itself. Knowing the conditions under which something fails is knowing the conditions under which you can trust it.

The Community Practice

Shared templates are necessary but not sufficient. What makes a community of practice work is iteration: someone uses a template, finds a condition where it breaks, posts the failure, the template gets revised, and the revision gets used by the next person.

That cycle requires a feedback loop with low friction. The community infrastructure for this is straightforward: a Gitea repository at `http://10.0.0.80:3000/hyp3rsoft/ai-for-dbas-community` (a separate repo from Bob itself), with prompt templates as structured YAML, contribution guidelines that focus on reproduction steps for failures rather than just success stories, and a tagging system that lets people filter by SQL Server version, model size, and failure class.

The harder part is cultural. Senior DBAs are the person with the answers. That identity makes posting failures hard. Saying "this prompt I thought was good produced a hallucinated recommendation in this specific case" requires a different professional orientation. It requires the framing that sharing failure information is a contribution to collective knowledge, not a confession of incompetence.

That framing is one I am actively trying to model. The chapter on the rollback-first pattern in this book exists because Bob applied a bad fix once before the snapshot requirement was enforced, and fixing the aftermath cost half a day. That story is in the book because the lesson is worth more than the embarrassment costs.

Why This Matters Beyond One Agent

The honest answer to "why should a DBA care about AI" is not about job preservation. It is about leverage.

The DBAs who will benefit most from this community are the ones who are already stretched too thin: one person managing twelve instances, on-call every other week, no one junior enough to delegate the routine work to. Bob, configured well with patterns from the community library, is not a replacement for that person. It is the person's ability to be in two places at once: monitoring the routine while the DBA focuses on the things that actually require judgment.

The pattern library is how one person's hard-won knowledge about what a `SOS_SCHEDULER_YIELD` spike looks like at 2 AM becomes something fifteen people can act on without earning it themselves. That is not democratization in the abstract sense. That is a specific, practical transfer of diagnostic expertise from the people who have it to the systems that can apply it at scale.

Chapter 10: Scaling Sovereignty

Bob started as a single-instance monitoring agent running in a home lab. The architecture was designed to be one person's answer to a recurring problem. What this chapter describes is what happens when that architecture meets the reality of a digital agency or a managed services provider with multiple clients, multiple regulated environments, and a mandate that none of those environments bleed into each other.

The technical changes are modest; the organizational ones are harder.

The Multi-Client Problem

A digital agency running SQL Server for enterprise clients faces a different version of the report problem than a single DBA monitoring one environment. It is not 214 pages. It is fourteen separate 214-page reports, for fourteen clients, each with its own schema, its own compliance requirements, its own definition of "critical." The problem is not volume per client. The problem is the sum across clients, and the fact that each client's data must be treated as if the other clients do not exist.

That last point is where cloud AI fails hardest. If you send client A's query plans to an external API for analysis, and the same API processes client B's data ten minutes later, you have a concrete data residency problem that shows up in GDPR data processing records, in PCI DSS scoping reviews, in SOC 2 control attestations. A shared external API is a shared data processor. Shared data processors require agreements, documentation, and in some regulatory regimes, prior client consent.

Local inference eliminates that problem by eliminating the shared processor. Client A's query plans and client B's query plans are processed by the same Ollama instance on your hardware, but they never leave your network, and they never commingle in any external system's memory. You are the data processor. You have always been the data processor. The AI did not change that.

This is the argument for local sovereignty at agency scale. It is not philosophical. It is the concrete answer to a compliance question your legal team will eventually ask.

The Architecture at Scale

Scaling Bob from one environment to multiple clients requires two changes to the existing architecture.

The first change is multi-tenancy in the database layer. The `BookOfBob` schema on `db-host` at `10.0.0.14:50003` is currently single-tenant: one set of tables, one history, one remediation log. For multi-client operation, you need either separate databases per client, which is clean but expensive in SQL Server licensing terms, or a tenant-aware schema that adds a `client_id` column to every table and enforces row-level security so the agent serving one client cannot accidentally read another client's history.

The row-level security approach is correct here. It is also exactly the kind of database security work a DBA should be comfortable with:

```
-- Add client isolation to the snapshot history table
ALTER TABLE dbo.SnapshotHistory
ADD client_id INT NOT NULL DEFAULT 0;

-- Create a security policy that filters rows by client context
CREATE FUNCTION dbo.fn_SecurityFilter(@client_id AS INT)
RETURNS TABLE
WITH SCHEMABINDING
AS
RETURN SELECT 1 AS result
WHERE @client_id = CAST(SESSION_CONTEXT(N'client_id') AS INT);
GO

CREATE SECURITY POLICY dbo.ClientIsolationPolicy
ADD FILTER PREDICATE dbo.fn_SecurityFilter(client_id) ON dbo.SnapshotHistory,
ADD FILTER PREDICATE dbo.fn_SecurityFilter(client_id) ON dbo.FindingHistory,
ADD FILTER PREDICATE dbo.fn_SecurityFilter(client_id) ON dbo.RemediationLog
WITH (STATE = ON);
GO
-- source: bob@c549c88
```

The agent sets the session context at connection time:

```
def get_client_connection(client_id: int) -> pyodbc.Connection:
    """
    Return a database connection scoped to a specific client.
    The session context enforces row-level security for the lifetime
    of this connection. Do not share connections across clients.
    """
    conn = pyodbc.connect(CONNECTION_STRING)
    conn.execute(
        "EXEC sp_set_session_context N'client_id', ?",
        client_id
    )
    return conn
# source: bob@c549c88
```

With this in place, the monitoring agent for client A physically cannot read client B's history, even if a bug in the routing logic passes the wrong connection. The database enforces the isolation.

The second change is agent configuration isolation. Each client needs its own system prompt context (their specific applications, their compliance requirements, their escalation contacts) and its own operational windows (when is it acceptable to auto-apply index maintenance for this client). The cleanest approach is one configuration file per client, pulled by the agent at the start of each monitoring cycle:

```
/etc/bob/clients/
    client-001.env
    client-002.env
    client-003.env
```

Each file contains the client-specific settings: SQL Server connection string, Proxmox VM ID for snapshots, operational window schedule, alert contacts, and the path to the client-specific system prompt addendum. The base system prompt is shared. The addendum is where client-specific knowledge lives: "this client's top-10 reports run at 6 AM every Monday, expect elevated CPU and do not alert on that pattern."

The Inference Layer Does Not Need to Scale

One common assumption when designing multi-client architectures is that you need more hardware per client. That assumption is wrong here.

A single `inference-host-1` at `10.0.0.70` with its RTX 3090 handles concurrent diagnostic requests through Ollama's `OLLAMA_NUM_PARALLEL=2` setting. In

practice, a monitoring cycle for one client takes between 2 and 8 seconds of inference time. With a 60-second probe cycle per client and up to a dozen clients, the inference load is easily within the capacity of a single GPU host running `gemma4:26b`.

The inference host does not know whose data it is processing. It receives a prompt, generates a response, returns it. The client isolation is handled entirely in the agent layer and the database layer. `inference-host-1` is a shared resource; the isolation boundary is upstream of it.

This is also why the data residency argument works. The LLM at `10.0.0.70` processes client data, but that data never leaves the `10.0.0.0/24` network. The LLM is your tool. It does not have a customer identifier, a telemetry endpoint, or a terms-of-service agreement that allows the provider to use your prompts for model training. The Ollama instance on your hardware is a tool, not a vendor relationship.

The Business Case

The business case for local sovereignty at agency scale is a cost and compliance argument that gets stronger the more clients you add.

On the cost side: cloud API pricing for high-frequency monitoring across twelve clients would run into meaningful money. The monitoring cycle for a busy SQL Server instance, sixty seconds, ten to fifteen minutes of wait stats, top-query data, and a query plan or two, is a several-hundred-token prompt. At current GPT-4 pricing, running that continuously across twelve clients would cost more than the hardware depreciation on `inference-host-1` in the first few months. After the hardware is paid off, the marginal cost per additional client is zero.

On the compliance side: the ability to tell a GDPR-regulated client in the EU that their database diagnostic data never leaves the LAN is not just a legal convenience. It is a sales differentiator. Most of the managed services providers competing in this space are using cloud tooling because it is easier to set up. Local sovereignty requires infrastructure investment and technical depth. It is not the path of least resistance, and that is exactly why it differentiates.

The combination of zero marginal inference cost and complete data residency control changes the calculation for which clients you can take on. Healthcare organizations under HIPAA, utilities under NERC CIP, financial services under PCI DSS: these are the clients with the most acute data residency requirements and the most willingness to pay for a provider who can credibly answer the compliance question. Cloud tooling cannot give them that answer. Local inference can.

The economics work out as follows. A monitoring cycle for one SQL Server instance, running every 60 seconds, generates a prompt of roughly 500 to 1,500

tokens depending on the diagnostic payload size. Call it 1,000 tokens per cycle as a conservative average. Sixty cycles per hour, 24 hours per day, 365 days per year, across twelve clients: that is roughly 6.3 billion tokens of input per year. At GPT-4 Turbo pricing as of 2025-01, input tokens cost $0.01 per 1,000. That is $63,000 per year in inference costs, before output tokens. Output tokens at $0.03 per 1,000 add roughly another $30,000 annually for typical diagnostic response lengths. Call the cloud API cost $90,000 per year to monitor twelve SQL Server instances continuously.

`inference-host-1` at `10.0.0.70` with the Ryzen 9 9950X and RTX 3090 cost approximately $3,500 in hardware. Power consumption running inference continuously is roughly 400 watts, or about 3,500 kWh per year. At $0.12 per kWh, that is $420 per year in electricity. Depreciate the hardware over three years: $1,167 per year. Total cost: roughly $1,600 per year to monitor twelve clients.

The break-even against cloud API costs happens in about a week. After the hardware is paid off, every additional client is nearly free to add. A 5-DBA shop choosing between cloud AI monitoring and local inference is choosing between a service that costs more per year than many of their client contracts are worth, and a system they own outright and can extend without asking permission.

The political angle of this math is the part that gets underweighted. The agency that owns its inference layer does not have a vendor relationship that can change pricing, deprecate a model, or decide to train on your clients' data as a terms-of-service update. Cloud AI vendors have done all three of those things in the last two years. Each time they do it, the agencies dependent on them have to respond: renegotiate contracts, update privacy policies, re-evaluate compliance posture. That reactive work costs time and attention that a local inference deployment does not require.

The agency that built Bob controls what version of the model it runs, when it upgrades, what data enters the inference context, and what pricing it charges clients. None of those decisions require a third party's approval or pricing team's roadmap. That independence is the point.

Where This Goes

Eighteen months of building has produced an architecture that works. One DBA's home lab, two inference hosts, one SQL Server target, one monitoring loop. The core is proven. The extension to multiple clients is a configuration and schema change, not an architectural one.

The next eighteen months are about proving the extension: deploying the multi-tenant configuration in the agency context, building the track record that justifies expanding the auto-apply scope, and making the community artifacts good enough that another DBA can get from zero to a running deployment without having to read the source code to understand what is happening.

That is not a moonshot. It is the natural progression of a system that was designed to be extended. The architecture supports it. The hardware is already running. The reasoning layer is proven. Extension is configuration.

What remains is the work of making it available to more than one server room.

The book ends here. The project does not.