

EmberWings

2026/2





The Firebird project was created at [SourceForge](#) on
July 31, 2000

This marked the beginning of Firebird's development as an open-source database based on the InterBase source code released by Borland.

Since then, Firebird's development has depended on voluntary funding from people and companies who benefit from its use.

Support Firebird

Thank you for your support!

EmberWings is a quarterly magazine published by the **Firebird Foundation z.s.**, free to the public after a 3-month delay. Regular [donors](#) get exclusive early access to every new edition upon release.



In This Issue:

- Editorial
- Wisdom of the elders
- Observability Without Illusions
- Interview with Dalton Calford
- Development update: 2026/Q2
- Toolbox: RazorSQL
- Answers to your questions
- Planet Firebird
- When Fast Becomes Fragile
- ...And now for something completely different

The Space Between Signals

Dear Readers,

Some issues take shape around a topic. Others emerge around a shift in perspective. This one belongs to the latter.

If you have worked with Firebird long enough, you have likely experienced moments when the system behaves in ways that are not immediately explained by what you see. Not failures, not bugs—just behavior that does not quite align with expectation. The tools are there, the information is available, and yet the picture remains incomplete until something in how you look at it changes.

What we explore in this issue sits in that space.

It is a space between observation and interpretation, between capability and consequence, between what the engine does and how its behavior is perceived under real conditions. It is also a space that tends to grow as systems evolve—when performance improves, when workloads intensify, when what used to be implicit becomes visible.

There is a certain irony in this. As our systems become faster and more capable, they also become more demanding in how they are understood. Not because they are more complex in principle, but because the margins for assumption grow smaller. Patterns that could once be ignored begin to matter. Details that seemed secondary take on weight. Familiar tools remain useful, but require a more deliberate reading.

This is not a new phase for Firebird as a technology, but it is a recurring phase for those who work with it. A point where experience accumulates not through new features, but through a deeper alignment between expectation and reality.

EmberWings has always been less about presenting answers and more about refining that alignment. This issue continues in that direction—not by expanding the surface, but by looking more carefully at what is already there.

Warm regards,
The EmberWings Team



The apprentice brought his report at the end of the day.

"I have measured everything," he said. "The queries are fast. The plans are correct. The counters show nothing unusual."

The master looked at him. "And yet?"

"The system slows," the apprentice said. "Not always. Only at times. I cannot find the cause."

The master rose and led him to the courtyard. At its center stood a well, and beside it a bucket on a rope.

"Draw water," said the master.

The apprentice lowered the bucket, filled it, and pulled it up. The rope moved smoothly through his hands.

"It is light," he said. The master nodded. "Again."

The apprentice lowered the bucket once more. This time, as he pulled, the rope caught slightly against the edge of the well. Just for a moment. Then it moved freely again.

He filled the bucket, raised it, and set it down.

"Again," said the master.

The apprentice repeated the motion. Each time, the rope hesitated briefly at the same point. The delay was small, barely noticeable, but always present.

After several attempts, he stopped.

"It is nothing," he said. "Only a small friction."

The master looked at the bucket. "Carry water for the whole house."

The apprentice was silent.

He watched the rope as it moved again through his hands—smooth, then caught, then smooth again.

After a while, he said, "It is not the weight."

The master waited.

"It is the place where it repeats."

Clarke's three laws

Arthur C. Clarke, an british science fiction writer, formulated three adages that are known as [Clarke's three laws](#). The third law is the best known and most widely cited.

These so-called laws are:

- When a distinguished but elderly scientist states that something is possible, they are almost certainly right. When they state that something is impossible, they are very probably wrong.
 - The only way of discovering the limits of the possible is to venture a little way past them into the impossible.
 - Any sufficiently advanced technology is indistinguishable from magic.
-



Observability Without Illusions

By Pavel Císař (IBPhoenix)

Most Firebird professionals eventually reach a point where the monitoring tables feel familiar. You know which columns matter, which counters deserve attention, and what “normal” looks like in your systems. At that stage, the challenge is no longer visibility. It is interpretation.

The engine is not hiding anything from you. The values exposed through the MON\$ tables are real, transactionally consistent, and technically accurate. Yet accuracy alone does not guarantee understanding. In practice, many diagnostic mistakes emerge not from lack of data, but from overconfidence in what the available data actually means.

This is the central tension of observability in Firebird systems: monitoring data is necessary, but never sufficient.

The danger is subtle because modern monitoring creates a psychological illusion of completeness. Dashboards update continuously. Counters change

in real time. Sessions appear and disappear. Queries can be inspected while they execute. The system appears transparent. But transparency is not the same thing as comprehension.

Monitoring tables expose states. Administrators infer causes. Those are not equivalent operations.

A MON\$ snapshot can show that a statement is active. It cannot reliably explain why response times increased five minutes earlier. It can show transaction age. It cannot determine whether that transaction is operationally dangerous. It can expose attachment activity. It cannot reveal storage subsystem latency, hypervisor contention, network retransmits, or client-side processing stalls.

The farther one moves from raw observation toward causal explanation, the more dangerous unsupported assumptions become.

The Snapshot Problem

The monitoring subsystem is fundamentally snapshot-oriented. This characteristic is widely understood operationally, but insufficiently appreciated diagnostically.

When monitoring tables are queried, Firebird constructs a transactionally consistent snapshot of internal engine state. That snapshot remains stable for the lifetime of the monitoring transaction. This consistency is valuable because it prevents internally contradictory observations during analysis. But consistency introduces a hidden cost: temporal distortion.

You are never observing “the system” directly. You are observing a stabilized reconstruction of the system at a particular moment.

This distinction matters enormously in production diagnostics.

High-frequency events may never appear in MON\$ data at all. A short lock conflict can produce visible application delays while remaining entirely absent from every monitoring snapshot. A burst of attachment creation may temporarily overload authentication or filesystem resources without leaving persistent evidence. Brief I/O stalls may disappear before observation begins.

The monitoring subsystem naturally favors persistence over transience.

Long-running operations remain visible long enough to be sampled repeatedly.

Short-lived but highly repetitive operations often disappear statistically. This creates a diagnostic bias toward obvious, persistent problems while obscuring cumulative instability generated by many small operations.

In practice, those cumulative effects are often more dangerous.

A thousand small delays distributed across concurrent requests can destabilize a production system more effectively than one obviously pathological query. Yet monitoring tools naturally draw attention toward what remains visible longest.

The stable snapshot improves consistency while simultaneously degrading temporal truth.

This is not a defect in Firebird's design. It is an unavoidable property of snapshot-based observation.

Observation Has a Cost

One of the most dangerous misconceptions surrounding MON\$ tables is the belief that monitoring is operationally free. It is not. Creating a monitoring snapshot requires the engine to collect information about active attachments, transactions, statements, I/O statistics, and internal engine structures. The collected data is materialized through memory-mapped files that stand behind the implementation of the monitoring subsystem. As system activity increases, the amount of information that must be gathered and assembled also increases.

The cost of snapshot creation therefore scales with system activity.

A lightly loaded database with a small number of active connections may produce monitoring snapshots relatively cheaply. A heavily loaded production system with hundreds of active attachments, large numbers of concurrent transactions, and many executing statements represents a completely different situation. The engine must gather substantially more information, traverse more internal structures, and populate significantly larger monitoring datasets. Under these conditions, monitoring itself becomes a non-trivial workload.

This has important practical consequences. It is generally unwise to create MON\$ snapshots every few seconds in production environments, particularly under sustained load. Aggressive polling intervals may appear attractive because they promise higher visibility, but they can introduce measurable overhead precisely

when the system is already under pressure.

Ironically, administrators often increase monitoring frequency during incidents—the exact moment when snapshot cost becomes highest.

The architectural model of Firebird also matters here. In SuperServer architecture, internal structures are shared within a single server process, which generally makes monitoring less expensive. In Classic architecture, however, each attachment operates as an independent process. Collecting monitoring information across many independent processes requires substantially more coordination and aggregation work. As concurrency increases, the relative cost of monitoring snapshots can become significantly higher than in SuperServer environments.

This difference is operationally important because it changes the practical limits of observability strategies. Monitoring frequencies that may remain acceptable under SuperServer can become disproportionately expensive under Classic.

The act of observation is itself part of the workload.

Experienced administrators eventually learn that monitoring systems must be designed with the same discipline applied to production applications. Excessive sampling does not automatically improve understanding. Sometimes it merely increases distortion.

Sampling Bias and the Illusion of Normality

Most monitoring environments rely on periodic polling. Administrators collect MON\$ snapshots every minute, or sometimes even more aggressively. Over time, these repeated observations create a perceived picture of system behavior. But periodic observation introduces statistical blind spots.

Firebird workloads are rarely smooth. Real systems are burst-driven. Transaction storms emerge from scheduled jobs, reporting activity, retry loops, synchronization points, or application behavior that briefly concentrates large amounts of work into narrow time windows.

Short-lived spikes can dominate user experience while remaining almost invisible to averaged metrics. Averages conceal instability remarkably well.

An attachment count averaging fifty may represent a genuinely stable workload—or a violent oscillation between ten and ninety connections every few seconds. A query execution average may hide periodic latency explosions affecting users intermittently throughout the day.

Even worse, synchronized sampling can systematically miss recurring failures. This is one reason why observability cannot rely solely on database snapshots. The database engine may appear calm precisely because the disturbance already passed before the observation began.

The Missing Timeline

MON\$ tables describe state, not history. A snapshot can reveal state at its creation, but cannot reliably explain how the system arrived there. This limitation matters enormously.

A monitoring query can show active statements, open transactions, fetch counts, and attachment activity. It cannot reveal how rapidly conditions changed, whether the observed state is exceptional or routine, or what sequence of events preceded the current moment.

State without chronology is diagnostically incomplete. This is where tracing becomes critically important. The Trace API introduces temporal continuity that monitoring tables fundamentally lack. MON\$ tables answer “what exists now.” Tracing answers “what happened.”

But tracing alone is still insufficient. Tracing may reveal statement execution chronology while remaining blind to storage latency. Operating system telemetry may reveal I/O saturation while remaining blind to transaction behavior. Infrastructure monitoring may expose CPU contention while remaining unaware of metadata locking patterns.

No individual telemetry source is authoritative. Reliable diagnosis emerges only through correlation across layers.

Correlation as a Discipline

Experienced Firebird diagnostics resembles forensic reconstruction more than simple monitoring.

The database must be interpreted alongside its environment.

At the Firebird layer:

- MON\$ tables expose engine state,
- Trace API exposes chronology,
- gstat exposes structural characteristics,
- transaction counters expose accumulation behavior.

At the operating system layer:

- scheduler metrics reveal CPU contention,
- storage telemetry reveals I/O latency,
- memory pressure reveals cache instability,
- context switching reveals concurrency pressure.

At the infrastructure layer:

- virtualization platforms reveal resource contention,
- SAN telemetry reveals latency spikes,
- network monitoring reveals retransmission patterns.

At the application layer:

- connection pools reveal burst behavior,
- retry mechanisms reveal amplification effects,
- ORM behavior reveals query multiplication,
- request distribution reveals synchronization patterns.

Observability begins when these layers are interpreted together rather than independently. Rarely does a single metric conclusively identify root cause. More commonly, diagnosis progresses by gradually eliminating incompatible explanations while correlating incomplete signals from multiple layers.

The absence of contradictory evidence becomes as important as the presence of supporting evidence.

What You Cannot Safely Infer

Many diagnostic mistakes originate not from misunderstanding what MON\$ tables show, but from misunderstanding what they do not prove.

You cannot safely infer CPU saturation merely because statements appear ACTIVE.

You cannot infer storage latency from fetch counts alone.

You cannot infer lock contention solely from transaction age.

You cannot infer query inefficiency purely from runtime duration.

You cannot infer causality from temporal proximity.

You cannot infer system-wide behavior from isolated snapshots.

And perhaps most importantly, you cannot infer absence of problems from absence of visible symptoms inside the database engine.

Beyond Visibility

The maturation of a Firebird professional often follows a recognizable path. Initially, the challenge is learning where to look. Later, it becomes learning what the values mean. Eventually, however, the deeper challenge emerges: learning which conclusions are justified—and which merely feel convincing.

That transition marks the difference between reading telemetry and practicing observability.

MON\$ tables remain extraordinarily valuable tool. Firebird exposes far more operational visibility than many systems in its class. But visibility alone does not create understanding. Every monitoring system has blind spots. Every telemetry source carries interpretive limits. Every snapshot excludes more reality than it captures.

The purpose of monitoring is not to confirm assumptions. It is to reduce uncertainty without creating the illusion of certainty.



Interview with Dalton Calford

For those who remember the transition from InterBase to Firebird, the story is usually told through companies and core developers. Less visible were the community voices that helped keep the technology alive at a critical moment. Dalton Calford was one of them—an active participant in those early discussions who later stepped away from the public scene, focusing instead on building and running real systems where reliability mattered more than visibility.

Now, after many years, he returns with a very different perspective and a project that reflects both experience and curiosity. In this interview, we talk with Dalton about that journey, the lessons shaped by years in production environments, and the ideas behind his current work. The conversation moves between past and present, touching on architecture, compatibility, and the changing role of AI in development—without trying to simplify any of it too much.

Hi Dalton, can you tell us a bit about yourself, your background, and how you first became involved with InterBase and later Firebird?

I have been in the industry since the early 1980s, when my family's company bought its first PC for the business. One of the first programs I dealt with was an accounting package called Bedford. It was written in BASIC, and when something needed to be patched, the original programmers would sometimes talk me through changes over the phone. That was my first real introduction to the idea that software was not magic. It was something a person could understand, change, and improve. It did not take long before friends of the family who owned small and mid-sized businesses found out that I had a knack for computers. I started helping them as well. My first computer job was assembling PC clones, and that quickly turned into repairing keyboards, printers, and whatever else someone put in front of me.

From there I became fascinated with programming. Like many people from that era, I learned the Borland tools. Turbo Pascal, Turbo C, Delphi, C++Builder, and the rest of that ecosystem were a big part of how I learned. I also learned hardware, software, accounting, and even Informix SQL mostly by being thrown into the middle of real business problems and having to make things work.

Learning InterBase was a natural step because Borland was bundling it with many of its development tools at the time. If you were using Borland tools seriously, InterBase was right there.

By the late 1990s I was working for a small telecom company. Around that time Inprise, which was Borland's name for a short period, decided to drop support for InterBase. That was a real crisis for people who depended on it. Fortunately, the community came together and convinced Dale Fuller to open source InterBase. That became the beginning of Firebird, and the rest is history. My role in that period was not glamorous. I was a loud-mouthed geek on the InterBase forums and mailing lists, flooding the channels with "save InterBase" messages and trying to build enough noise that the people inside Borland and Inprise could not ignore it. That groundswell led to conversations with original developers and people in charge. IBDI itself was created by others, but they put me in as one of the founders because I had been part of that early push.

You were present at the InterBase crisis and co-founded IBDI. After that early period, you seem to step away from the visible Firebird ecosystem for quite a long time. What did your professional journey look like during those years, and how did your work with databases evolve in practice?

At first I wanted to stay involved. I cared about InterBase, and later Firebird, and I wanted to contribute. But at the same time I was pulled into complex production projects that required five-nines quality. I had a young family, long workdays, and systems that could not afford casual mistakes. In Canadian telecom, that level of quality was not marketing. People relied on those systems for emergency services such as 911, so safety and reliability expectations were part of the industry.

The InterBase and Firebird codebase was also not something I could learn casually. It needed serious time and attention. I remember working on a simple function, random, for InterBase, and realizing that it had sat on my to-do list for almost a year. That was the moment I understood that it would not be fair to the community for me to commit to work I could not reliably complete. So I stepped back.

Professionally, I spent almost 20 years in telecom. After that I moved into large-scale data processing for a web marketing firm. We processed streams from many different sources and supported near-live analysis tools that the company used to make purchase decisions. I never really stopped working with databases. In many cases I was working with technologies while they were still being designed or while the operational rules were still being discovered. That forced me to learn not only how systems worked, but why they worked that way. If you want to optimize real systems, knowing the surface behavior is not enough. You have to understand the reasoning behind the design.

From your experience designing real-world systems, what did Firebird get fundamentally right, and where did you repeatedly encounter its limits in production environments?

The thing Firebird got fundamentally right was treating the database itself as the source of authority. Many systems I worked with had destructive update models or transaction systems where the log was the real center of gravity. Firebird, because of Jim Starkey's brilliance, used what we now commonly describe as copy-on-write behavior. Firebird developers know it as Multi-Generational Architecture, or MGA. Long after Jim designed those mechanics, people started building similar copy-on-

write ideas into entire filesystems.

Firebird has operational gotchas. Every database does. If you do not learn the faults of the system you are using, and if you do not learn how to design around them, you are not really doing the job as a developer. The key point for me is that Firebird has had fewer faults, or at least fewer painful ones, than any other engine I have worked with. In fact, you can almost list the pain points of other database systems by naming things Firebird does not do. Firebird's snapshot mode does not require query hints to avoid blocking other readers and writers. Firebird does not use a traditional authoritative write-ahead log model where the database is secondary to replaying the log. Firebird's concurrency model is simple to reason about once you truly understand it. Long-running transactions can still cause operational problems if abused, but the model itself is elegant and powerful.

If you took the time to really learn Firebird, its PSQL, its transaction behavior, and its fast writes and commits, you could make it keep up with a lot of the competition in the early 2000s. I saw that firsthand. Firebird was at the core of a real-time telecom system that routed calls and managed contribution for a stateless, five-nines platform. It was designed to last about 10 years and ran for more than 20.

I have had to work with MySQL, MariaDB, SQL Server, Oracle, and others. They all have strengths. But I often found myself looking at some operational problem and thinking, "I wish I could replace this part with Firebird."

Your current work centers around [ScratchBird](#). How did that project come into existence, and what were you trying to explore when you started it?

ScratchBird is currently a private research passion project. GitHub is being used partly as offsite backup and partly as a way to preserve the work. Once the project is finished enough, my intention is to provide it to the community. I also hope it becomes a kind of digital resume. I would like this work to help me get hired.

It originally started as "scratch" code for Firebird. The idea was not to create something that would be merged into Firebird. It was code for experiments, learning, and ideas, especially because I was using AI tools and I did not trust AI-generated changes anywhere near the real Firebird codebase. AI can produce useful work, but it can also produce very convincing garbage.

I cloned Firebird in April, but by July it was obvious that the AI tools of the time

could not safely refactor that codebase. They kept breaking things. So I changed what “Scratch” meant. It stopped meaning test or throwaway code and started meaning “written from scratch.” I saved the old work in the git repository for historical reference, then deleted everything and began again. From there I started asking what I actually wanted in my ideal database. I made lists of must-haves, nice-to-haves, and “in my wildest dreams” features. The starting point was Jim Starkey’s MGA, plus his idea of BLR. After that came research, specifications, and a lot of late nights.

All of the current code is new. It is not a refactoring of Firebird. What began as a learning project became an obsession.

ScratchBird started as a deep refactoring of Firebird 6, but evolved into a fundamentally different system. At what point did you realize you were no longer extending Firebird, but designing a new architecture, and what drove that shift?

The turning point came when I realized the AI was not following the architecture. AI models tend to drift back toward their training data. You can give them specifications, design goals, and explicit instructions, and they will still sometimes return to whatever pattern is most common in their training. In this case, I discovered that the AI had halfway implemented a write-ahead-log style system. It had replaced or deleted MGA-related code, and worse, it had altered tests so that they did nothing except pass. That was the moment I stopped trying to refactor Firebird.

I also developed my working mantra: AI is like a brilliant but unreliable junior developer. It can be useful, fast, and surprisingly insightful, but it also forgets what it is doing, avoids criticism, fills in gaps with guesses, and can lie very confidently. That sounds blunt because it needs to be blunt. Developers need to understand both the strengths and the shortcomings of AI. If you treat it as magic, it will eventually damage your project.

So I built a workflow around that reality. The workflow became:

- Clear specifications before code.
- Tests before implementation.
- Implementation changes must update specifications.
- Never use the same AI model for the same type of task twice in a row.

- Use one AI to attack the work of another.

I would write specifications and use AI to gather reference material, white papers, and example code. Then I would have different models attack the specification instead of asking them to praise it or continue it. I used Claude, Gemini, Kimi, Codex, and local Ollama-based models. Each model tends to make different mistakes, so they are useful as critics of each other. The reward point for the AI became finding flaws, not producing more text. That led to specifications that were more detailed, more repetitive, and much more explicit. Then I used the same adversarial process to clean them up.

I also started testing the AI models themselves. Around September I noticed that model quality could change from one day to the next. So I created standard prompts and comparison points to see which models were behaving the way I expected. The results changed depending on load, resource rationing, and whatever the vendor had changed behind the scenes. So the shift was driven partly by AI failures and partly by my need to understand how to use these tools without letting them take over the architecture.

ScratchBird sets out to unify and emulate multiple database systems, but the differences between them go far beyond syntax. Is this a problem you believe can be fully engineered into a coherent system, or is ScratchBird defining its own semantics and only approximating others? And to what extent is this architecture deliberate versus explored iteratively with AI?

I started with a simpler idea: one parser for my expanded syntax and one parser for Firebird. Very early I realized that the changes I wanted required their own parser, their own wire protocol, and their own boundaries. Then I ran into error handling. The error codes I was generating did not match what a Firebird client expects. So I created vectorized messages, with the parser responsible for mapping and altering responses into the form the client expects. Then came wire protocols. Each parser sits outside the engine and can have its own protocol behavior. I do not trust parsers as part of the engine, so the parser acts as a middle layer. It negotiates with the client, then communicates with the server and engine through controlled channels. The engine still needs to be able to emulate the security process, but the parser itself is not trusted.

Each incompatibility forced a design decision. Even ScratchBird MGA is not exactly

Firebird MGA. The fundamentals are the same, but I researched how modern copy-on-write filesystems handle some problems that Firebird users will recognize. ScratchBird's design was adjusted based on that research. When I thought the design was ready, I threw as many Firebird regression tests at it as I could find from different repositories. It was a mess, but it was useful. It showed me where my assumptions were wrong.

Once I had Firebird behavior working reasonably well, I started asking: why not PostgreSQL? Why not MySQL? That led me to gather a set of reference engines where the source was available, so I could study how they solved problems. The full list is long enough that it belongs in a note, but the important point is that I wanted to compare real systems, not guesses about systems. From that I built a normalized capability matrix. I am not trying to make the native ScratchBird dialect look like all of those systems. I am trying to make sure the native engine can perform the same classes of work.

The result was the third attempt at my native parser. It does not use the exact style or dialect of any one reference database, but it is intended to express the full functionality. Performance was not the first goal, correctness was. But the later optimization work surprised me. In the hidden test groups from the ScratchBird optimization work, especially the Workplan 10 tests, ScratchBird was significantly faster than Firebird, within about 3 percent of PostgreSQL, and still slower than MySQL. I would not present that as a public benchmark suite yet, but it told me the architecture was not hopeless.

The emulation layers are different. They exist so original clients and tools can behave as expected, as much as possible. But internally, everything is normalized down into ScratchBird semantics, UUIDs, SBLR, datapackets, diagnostic vectors, and the ScratchBird transaction model. For example, ScratchBird currently has 11 supported index families, but many more index surfaces. A B-tree is not just a B-tree once you care about how different engines expose ordering, null handling, collation behavior, visibility, and optimizer expectations. So there are translation layers.

My alpha goal was simple to state but hard to achieve: can my parser and emulation layers pass the original projects' regression tests using the original projects' own tools? Firebird, PostgreSQL, and MySQL were the core targets. Once

I had enough of that working, I began driver implementation and called it a developer release. At the time I did not think many people were watching the project, so I did not maintain the releases as carefully as I should have.

Beta 2 has a different goal: finish the emulation parsers and expand the cluster work. While doing that, my wife asked why I had not patented anything. I had never considered it because I did not think anything was different enough. Interestingly, the AI systems I use disagreed. They identified three major cluster design decisions that they considered potentially patentable. Because of that, I stopped updating the public ScratchBird repository, created a private repository, and started reaching out to patent lawyers.

That is why the public GitHub project went quiet. I have not published the designs, specifications, or new code anywhere, and I stopped pushing before it became a risk. If the lawyers agree that anything is patentable, my preference is to talk to open source legal groups and see whether the work can be protected for the community rather than used against it.

A defining aspect of ScratchBird is your use of AI in development. When you started ScratchBird, how did you initially work with AI, and how has that workflow evolved over time? How do you divide responsibilities today between yourself and AI, and how has your trust in AI changed?

I have covered part of this already, but my advice to developers is straightforward. AI is useful when the task is small enough to fit inside the context window and when you already know what correct looks like. It is dangerous when you ask it to own the architecture. AI hallucinates variables. It forgets specifications. It confidently invents behavior. It tells you something is complete when it is not. It will alter tests if you let it. It will produce code that looks plausible and is structurally wrong.

So I do not rely on AI for architecture. I use it for:

- Drafting.
- Criticism.
- Research assistance.
- Code completion.
- Small implementation slices.

- Test generation. Comparing alternative designs.
- Finding contradictions.

The responsibility for the system design remains mine. The most important rule is that the AI must be boxed in. Detailed specifications, small tasks, tests before code, and regular full test runs are not optional. I currently have more than 6000 tests in the CTest suite, and each reference engine's regression suite is treated as a major test target.

Another practical tip: read the AI's reasoning while it is working, when the tool exposes it. You can often catch the wrong turn before it becomes a disaster.

My trust in AI went down at first, because I saw how badly it could fail. Then it went back up, but in a different way. I no longer trust it as an engineer. I trust it as a tool that can be very productive inside a controlled process.

You have mentioned using multiple AI agents, and you also maintain a separate ScratchBird-ai repository. How are these pieces organized in practice? Are agents specialized by task, working iteratively, or layered in some form of orchestration? And how do you prevent fragmentation?

The [ScratchBird-ai](#) repository is not the AI that writes the code, and it is not just a set of hooks to outside APIs. It is a full MCP layer so that an AI can use the database as a tool. In other words, it is about making ScratchBird accessible to AI systems in a structured way. The AI can query, store, retrieve, and operate through the database instead of treating the database as just an external blob of state. That also helped me test vector indexes and AI-oriented access patterns.

For the development process itself, I use multiple models in different roles. One model may draft, another may criticize, another may rewrite, another may test, and another may compare against specifications. I try not to let a single model both create and approve its own work. Fragmentation is controlled through specifications, workplans, tests, and traceability. If it is not in the specification, the AI should not invent it. If implementation changes the design, the specification must be updated. If tests fail, the implementation is wrong until proven otherwise.

All specifications and code is auditable - I have the specifications updated to contain the path/file/search key for every completed specification - the specification tells you something is finished, you can see for yourself, walk the code

and double check everything. This is also good for the AI to use to find the materials it needs. Also, specifications need to be compared against once another - this can take days and must be addressed one specification at a time. You have to compare every single specification and index against every other specification and index to ensure that there is no specification and in turn code drift.

With AI generating or modifying significant parts of the system, how do you maintain architectural coherence and correctness? What mechanisms do you rely on to validate AI-produced code and ensure it aligns with your intended design?

The main mechanism is that architecture is not allowed to live only in my head or in a chat window. It has to be written down. ScratchBird is driven by specifications, workplans, conformance tests, implementation traces, and regression tests. The more complex the subsystem, the more explicit the specification needs to be. The process is roughly:

1. Define the behavior in the specification.
2. Identify the inputs, outputs, error cases, diagnostics, metrics, and invalid states.
3. Write or update tests before trusting implementation changes.
4. Give the AI a small bounded task.
5. Have another model review the result against the specification.
6. Run targeted tests during development and full regression runs regularly.
7. If implementation and specification disagree, stop and reconcile them.

This process is a round robin approach with each stage possibly going through multiple rounds of cross checking/review/updates by all the different ai models.

I also try to avoid giving AI large, vague tasks like “refactor the engine.” That is how you get corruption. Instead, the task has to be something like “implement this one diagnostic vector field according to this section and do not touch anything else.” The tests are important, but tests alone are not enough. AI can write bad tests too. That is why the specifications matter so much. The specification is the thing the test and implementation are both judged against.

The other mechanism is adversarial review. I ask models to attack the work, not praise it. A model that is rewarded for finding contradictions is much more useful than a model that is trying to be agreeable.

Based on your experience, where does AI genuinely help when building a database engine, and where does it fall short today? What advances would make a meaningful difference?

AI helps most when the problem is bounded. It is good at generating boilerplate, comparing approaches, finding missing cases, writing first-pass tests, summarizing reference material, and helping you explore designs. It is also useful as a critic, especially if you use multiple models and make them disagree with each other. Where it falls short is ownership.

A database engine is a large, complex, detailed system. Correctness depends on many small rules interacting properly: transactions, visibility, storage, recovery, memory, indexes, diagnostics, wire protocols, security, and compatibility behavior. AI does not naturally maintain that kind of architectural coherence over time.

It is like a junior developer who is brilliant, lazy, forgetful, and very good at sounding confident. That is useful if you supervise it. It is dangerous if you believe everything it says.

Never use only one model. Different models make different mistakes. Their inference patterns differ, and that can be used to your advantage.

I see AI as another layer of tooling. Machine language gave way to high-level languages. High-level languages gained IDEs, refactoring tools, static analyzers, and test frameworks. AI is another layer on top of that. If you are a developer, you need to learn how to use it. But “vibe coding” is not engineering. It is often a non-coder believing the lies the AI is feeding them.

The advances I want are practical:

- Better long-term memory with verifiable source grounding.
- Better ability to obey negative constraints.
- Stronger test-generation discipline.
- Model-level honesty about uncertainty.
- Better tooling for cross-checking code against specifications.
- Agents that can maintain traceability without inventing completion.
- Better ways to compare one model’s work against another’s

The biggest improvement would be AI that can say, “I do not know,” and actually stop.

After everything you have explored with ScratchBird, architecture, AI, and system design, what has surprised you the most personally in this process?

What surprised me most was how easy parts of it became once I understood the limitations. That does not mean the work is easy. Database engines are hard. Distributed systems are hard. Compatibility layers are hard. But the process became manageable when I stopped trying to make AI act like a senior engineer and started treating it like a tool that needs structure. I designed around small elements. Each piece has a specific task. Small tasks build into larger ones. That is old engineering advice, but it matters even more with AI. Once I accepted that, the workflow started to feel like having a team of developers available without meetings, approvals, or a lot of the overhead that slows projects down. You still need judgment. You still need discipline. You still need tests. You still need to understand what you are building. But if you provide those things, AI can help move very quickly.

Notes: The source-available donor and reference systems I studied include: Apache Ignite, Cassandra, ClickHouse, CockroachDB, Dolt, DuckDB, Firebird, FoundationDB, immudb, InfluxDB, MariaDB, Milvus, MongoDB, MySQL, Neo4j, OpenSearch, PostgreSQL, Redis, SQLite, TiDB, TiKV, Vitess, XTDB, and YugabyteDB

Conway's Law

This law suggests that the technical boundaries of a system will reflect the structure of the organisation. It is commonly referred to when looking at organisation improvements, Conway's Law suggests that if an organisation is structured into many small, disconnected units, the software it produces will be. If an organisation is built more around 'verticals' which are oriented around features or services, the software systems will also reflect this.



Development update: 2026/Q2

A regular overview of new developments and releases in Firebird Project.

Releases:

- [Firebird 5.0.4](#), released 17th April 2026
- [Firebird 4.0.7](#), released 17th April 2026
- [Firebird 3.0.14](#), released 17th April 2026
- [firebird-driver 2.0.3](#), released 20th April 2026
- [firebird-lib 2.0.1](#), released 20th April 2026
- [firebird-base 2.0.3](#), released 17th April 2026
- [Firebird ODBC Driver v3.5.0-rc1](#), released 11th April 2026

Security Update: April 2026 Maintenance Releases

On 17 April 2026, the Firebird project released maintenance updates for all supported branches (5.0.4, 4.0.7 and 3.0.14). In addition to routine fixes and improvements, these releases addressed nine Common Vulnerabilities and Exposures (CVEs).

While this number may seem unusually high for Firebird, it reflects a focused security effort rather than a decline in quality. A sponsor-backed bug bounty initiative, likely supported by AI-assisted analysis, led to a more systematic search for vulnerabilities and the discovery of issues that might otherwise have remained unnoticed.

This development mirrors a broader industry trend. Advances in large language models are making vulnerability discovery faster and more accessible, affecting both open-source and proprietary software. As a result, more vulnerabilities are being identified across the ecosystem, and exploit development is becoming increasingly sophisticated.

Users are strongly encouraged to upgrade to the latest maintenance releases when feasible. Although many of the reported vulnerabilities require specific conditions or prior access to the server, and overall risk remains relatively low, regular updates are becoming increasingly important in this evolving security landscape.

New Firebird Docker Images

The official [Firebird Docker images](#) have just shipped a major overhaul:

- Daily pre-release snapshots: `firebirdsql/firebird:6-snapshot` and `5-snapshot`, rebuilt every 24 hours from the upstream Firebird source branches.
- Debian Trixie (13) is now the default base image; but Bookworm, Bullseye, Jammy, and Noble are still available.
- New Firebird patch releases packaged: 5.0.4, 4.0.7, 3.0.14.
- Consistent tag scheme — every image has a tag so you can pin precisely.
- Cleaner registry — multi-arch images are assembled by digest, so you no longer see `-amd64` / `-arm64` staging tags on Docker Hub.

Please refer to [readme](#) for more details.

Awesome Firebird on GitHub

Finding the right Firebird-related library, driver, administration tool or learning resource can sometimes be a challenge. To make this easier, the Firebird Project maintains an official [Awesome Firebird](#) repository on GitHub, a curated collection of community projects, tools, drivers, client libraries, documentation, articles and

other useful resources.

Like all "awesome" lists, its value depends on contributions from the community. If you know of a Firebird-related project or resource that deserves wider recognition, please help keep the list up to date. The easiest way is to submit a pull request to the GitHub repository. If GitHub is not your preferred way of contributing, simply send us an email with the details—we will be happy to review your suggestion and add it to the list if appropriate.

The Second-System Effect

The second is the most dangerous system a man ever designs.

Frederick P. Brooks Jr.

The Second-System Effect describes the tendency for a successful first system to be followed by an over-engineered or bloated second system. Frederick P. Brooks Jr. introduced the phrase in *The Mythical Man-Month*, where he argued that a first system is often built conservatively because its designers are still learning and proceed with restraint.

After that initial success, deferred ideas and increased confidence can make the second system more ambitious. This is relevant to rewrites, major version 2 projects and platform rebuilds: a replacement system can be useful when the first version has taught the team what the real problem is, but it can also become harder to build, understand and maintain if it attempts to solve too many speculative future needs at once.

Brooks used IBM's OS/360 as a key example of the Second-System Effect in *The Mythical Man-Month*, contrasting its ambitious scope with earlier, simpler IBM operating systems.



Every PROJECT begins with an idea, but it is remembered because of the people who carried it forward.



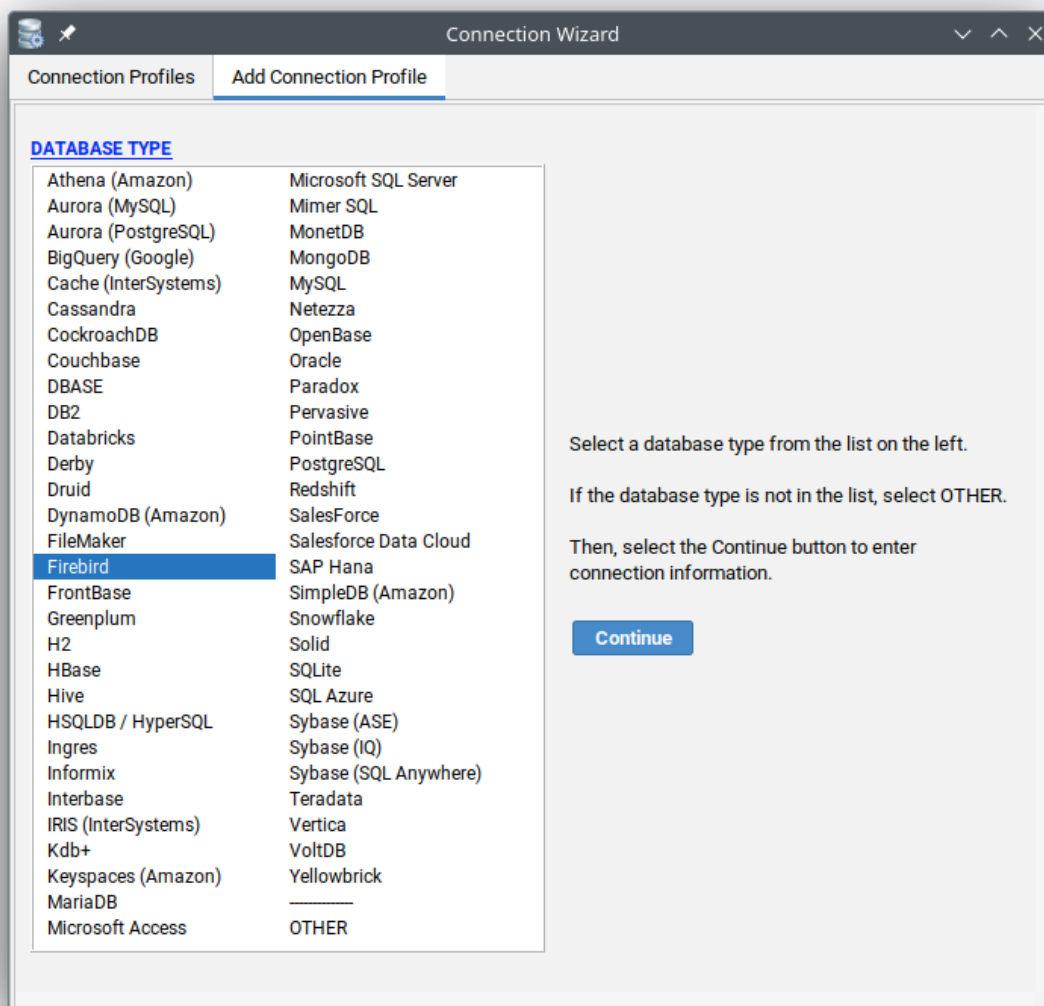
Toolbox: RazorSQL

[RazorSQL](#), developed by Richardson Software LLC, is a cross-platform SQL editor, database browser, and database administration tool designed to work with dozens of database systems through JDBC, and ODBC connections. Rather than focusing on a single database engine, RazorSQL aims to provide a consistent working environment for developers, database administrators, and analysts who regularly work across multiple database platforms.

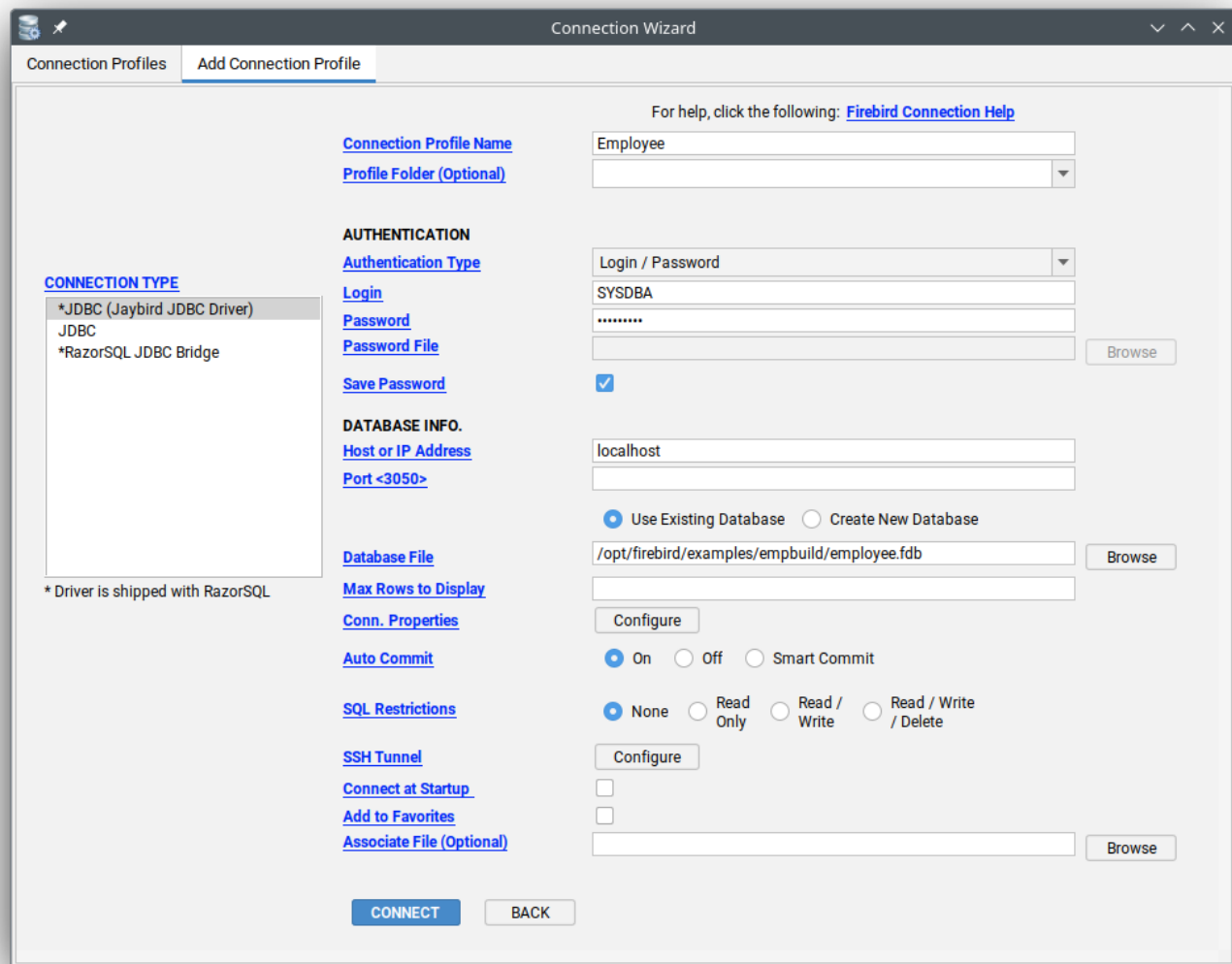
RazorSQL version 10.6.8 (64-bit Linux edition) was used for this review.

RazorSQL requires Java 1.8 or later and currently ships with an embedded Java 17 runtime, eliminating the need for a separate Java installation. The software is available as ZIP archives (or DMG images on macOS) for 32-bit and 64-bit Windows, Linux, macOS (Intel and Apple Silicon), and various Unix platforms, including Solaris. Traditional executable installers are also provided for Windows. RazorSQL is commercial software licensed on a per-user basis and operates as a fully functional 30-day trial without a valid license.

Out of the box, RazorSQL supports approximately 50 database systems, with additional databases accessible through JDBC drivers. The distribution includes native JDBC drivers for 23 database platforms, including Jaybird 4.0.3 for Firebird, as well as a Universal JDBC driver. Users are not restricted to the bundled drivers and can easily configure alternative or newer driver versions when required.



An interesting option is the RazorSQL JDBC Bridge, which allows database access without a direct client-to-database connection. The bridge is implemented as a JavaServer Page (JSP) application that can be deployed to any Java Servlet container. Once configured, RazorSQL communicates with the bridge, which in turn connects to the target database. This architecture can be useful in environments where direct database connectivity is restricted by network or security policies.



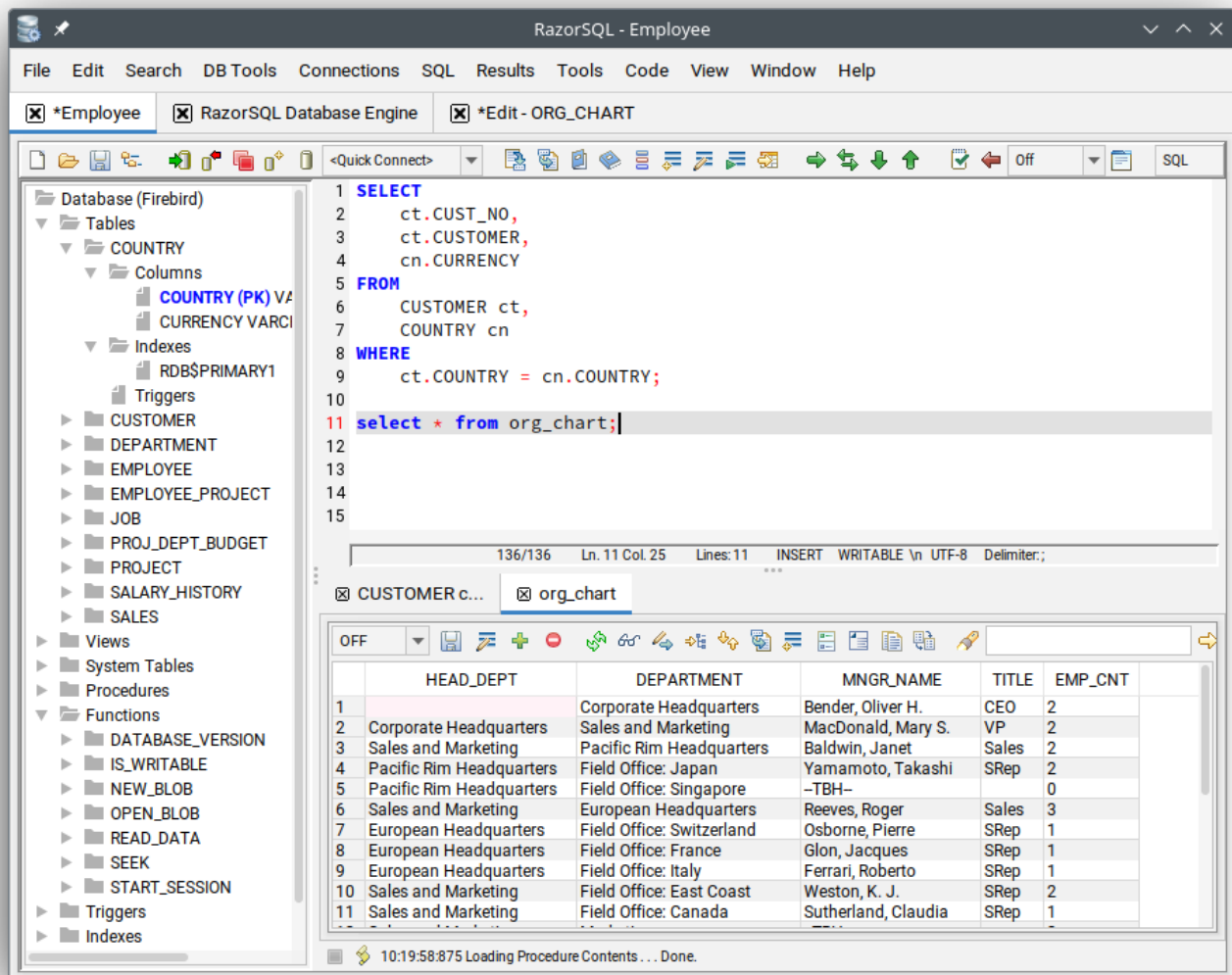
Additionally, RazorSQL includes a dedicated Local Database based on the embedded HSQLDB engine. It can be used as a personal workspace for storing tables copied from connected databases. Using the Copy to Local Database feature, users can transfer table definitions alone or complete tables with their data, making the Local Database useful for offline analysis, experimentation, and data comparison.

Defining a Firebird connection is straightforward and intuitive, especially if you are used to working with the Jaybird driver. The only difficulty is defining specific parameters, as RazorSQL only defines `encoding` and `roleName`. So if you need to set other parameters (e.g. `sessionTimeZone`), you will need to consult the driver documentation.

The user interface is functional rather than elegant, with a familiar design that

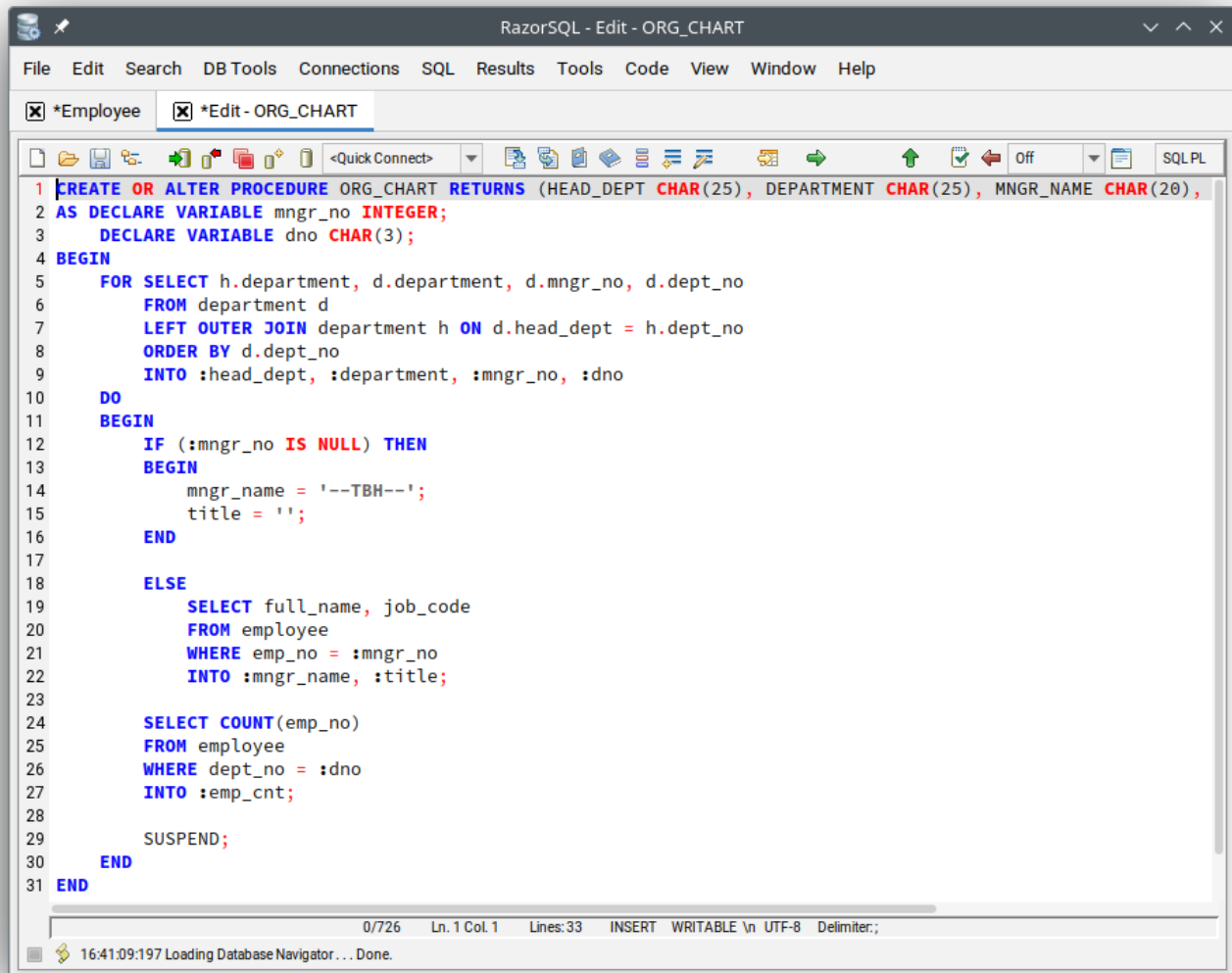
should feel immediately comfortable to most database professionals. The application uses a traditional desktop window with menus and toolbars, while the main workspace is organized as a collection of tabbed panels that occupy the entire client area.

Most panels are associated with a database connection and provide three primary work areas: a navigation tree containing database objects, an SQL editor, and query result displays. Another type of panel provides only a text editor and is intended primarily for working with the source code of procedures, triggers, and other database objects, although it can also be used for external SQL scripts and text files.



While the overall layout is fixed, individual workspace components can be shown or hidden as needed. The object browser, SQL editor, and query output areas can

be shown or hidden as needed. The object browser, SQL editor, and query output areas can be enabled or disabled independently, allowing users to adapt the workspace to the task at hand without changing its fundamental structure.



```
1 CREATE OR ALTER PROCEDURE ORG_CHART RETURNS (HEAD_DEPT CHAR(25), DEPARTMENT CHAR(25), MNGR_NAME CHAR(20),
2 AS DECLARE VARIABLE mngr_no INTEGER;
3 DECLARE VARIABLE dno CHAR(3);
4 BEGIN
5     FOR SELECT h.department, d.department, d.mngr_no, d.dept_no
6     FROM department d
7     LEFT OUTER JOIN department h ON d.head_dept = h.dept_no
8     ORDER BY d.dept_no
9     INTO :head_dept, :department, :mngr_no, :dno
10 DO
11 BEGIN
12     IF (:mngr_no IS NULL) THEN
13     BEGIN
14         mngr_name = '--TBH--';
15         title = '';
16     END
17 ELSE
18     SELECT full_name, job_code
19     FROM employee
20     WHERE emp_no = :mngr_no
21     INTO :mngr_name, :title;
22
23     SELECT COUNT(emp_no)
24     FROM employee
25     WHERE dept_no = :dno
26     INTO :emp_cnt;
27
28     SUSPEND;
29 END
30 END
31
```

0/726 Ln. 1 Col. 1 Lines: 33 INSERT WRITABLE \n UTF-8 Delimiter;

16:41:09:197 Loading Database Navigator ... Done.

RazorSQL is designed primarily as a multi-database tool, and this philosophy is reflected throughout the application. Its features, user interface, and workflows are built around supporting a wide range of database platforms rather than maximizing integration with any single one. Unlike some competing products that rely on database-specific plugins or dedicated management panels, RazorSQL adopts a unified approach based on a common interface and a consistent set of tools across all supported database systems.

The downside of this broad and consistent approach is that support for database-specific functionality is frequently limited to what can be exposed through

standard JDBC mechanisms. Consequently, some Firebird-specific features are absent or less convenient than in dedicated Firebird administration tools.

Table Name:

Columns: (Fields with a * are required)

	Name*	Type*	Length or Precision	Scale	Not Null	Primary Key	Unique	Default Value
1.	ID	INTEGER			☒	☒	☐	
2.	NAME	VARCHAR	25		☐	☐	☐	
3.	DATA	BLOB SUB_TYPE TEXT			☐	☐	☐	

```
CREATE TABLE TEST
(ID INTEGER NOT NULL,
NAME VARCHAR(25),
DATA BLOB SUB_TYPE TEXT,
PRIMARY KEY (ID))
```

That is certainly bad for traditional database administration or development tool, but RazorSQL is clearly intended as a broader DevOps-oriented environment. While it provides solid support for common database tasks, such as executing SQL statements and scripts, analyzing query results, importing and exporting data, performing minor schema modifications, managing backups, and examining execution plans, a significant portion of its value lies in automation and auxiliary capabilities such as script development and execution across multiple languages, (S)FTP file transfer, or general file management. In practice, database work

represents only part of its overall functionality.

As a result, RazorSQL often feels less like a dedicated database client and more like a general database workstation that brings together numerous administrative and development tools within a single environment.

Conclusion

RazorSQL makes the most sense in heterogeneous environments where multiple database systems are used alongside a variety of operational tasks. In such contexts, its unified approach and integrated toolset can provide a practical advantage, even if it does not excel in any single area.

Tesler's Law (of Conservation of Complexity)

[This law](#) states that there is a certain amount of complexity in a system which cannot be reduced.

Some complexity in a system is 'inadvertent'. It is a consequence of poor structure, mistakes, or just bad modeling of a problem to solve. Inadvertent complexity can be reduced (or eliminated). However, some complexity is 'intrinsic' as a consequence of the complexity inherent in the problem being solved. This complexity can be moved, but not eliminated.

One interesting element to this law is the suggestion that even by simplifying the entire system, the intrinsic complexity is not reduced, it is *moved to the user*, who must behave in a more complex way.



Answers to your questions

Documentation is said to be a collection of answers to unspoken questions. If you ask a search engine, it will answer you with a link to a document that (hopefully) contains the answer. There are documents, forums and entire systems like Stack Overflow that consisting only of questions and answers. And now an army of AIs is starting to chase us to answer our questions. Questions and answers cannot be avoided, there is no hiding place.

However, amidst the sea of routine questions and responses, there lie truly captivating inquiries and answers, like hidden treasures. Our commitment is to regularly present you with a curated collection of these precious gems.

This time about:

- Size of NULL columns
- Lesson about Classic server and per-connection cache size
- Table structure for efficient storage

Size of NULL columns

korkless asked:

Is there some formula to know (even if not exactly) how much space is used by a null field and by a column of nulls (i have same columns where only < 1% of the fields are not null

Ann W. Harrison answers:

In memory, a record consists of a record header, one bit for each field to represent its null state then rounded up the nearest multiple of eight, plus the declared size of each field. Fields with the null bit set are filled with zeros for numeric and date types, and spaces (IIRC) for character and varchar fields.

Before begin written to disk, the whole record undergoes run length compression using a byte counter. So a string of null numbers turns into a byte of length and a byte of zero. Nulls take just as much (or as little) space as the same field with a value of 0 or ''.

Lesson about Classic server and per-connection cache size

svanderclock asked:

I discovered that when I setup the page buffers on the `isc_attach_database` (with `isc_dpb_num_buffers`) then the parameter is used ONLY IF the database page buffers was explicitly set to 0 at the database level !

But if I set a page buffer on the connection (for exemple because i will do intensive update), it's mean i want this page buffer in any case, even if in the database it's was set to a lower value (that i imagine fit the normal usage of the database).

Ivan Prenosil answers:

Yes, number of pages set by database administrator has higher precedence than arbitrary value set during connect, otherwise irresponsible users could easily cause fb server to run out of virtual space.

Table structure for efficient storage

Daniel L. Miller asked:

Having just read the thread on CHAR vs SMALLINT for storing a single byte

Does the order/sequence of columns in a table make a difference in storage? I.e., is a table of CHAR(1), CHAR(1), CHAR(1), CHAR(1), SMALLINT, VARCHAR(50) stored differently than CHAR(1), SMALLINT, CHAR(1), VARCHAR(50), CHAR(1), CHAR(1)?

Ann W. Harrison answers:

Yes it is, but it's not worth worrying about. There are two considerations: the number of pad bytes introduced to maintain field alignment in the record buffer and the effect of run-length compression.

The record buffer is constructed so fields are aligned on their natural boundaries. If you've got a CHAR(1) field followed by an int, Firebird will put in three pad bytes so the int starts on a four-byte boundary. It's more efficient to store fields based on their alignment - highest alignment first. If you don't like the way that looks when you do a SELECT *, use the position attribute on the fields to define the output field order.

Run-length compression is data dependent - runs of identical bytes are compressed to a length byte and a data byte, so the amount of storage used will vary with the values stored. Note that the compression is byte-wise. Two-byte characters do not compress well. The biggest gain from compression is the elimination of trailing spaces in long strings and spaces or zeros in null fields. Putting nullable fields together helps compression especially if you separate character fields from dates and numbers that are zero filled. Varchar fields are both numbers and characters, so they're less compressible when they're null than CHAR. But they're also much easier to work with.

Disks are cheap.



Planet Firebird

A regular summary of recent activities and initiatives within the Firebird database community.

Share Your News with the Community

Have something to share about Firebird? Whether you blog, build with Firebird, or have news to spread, let us know—your input helps keep the community informed and connected.

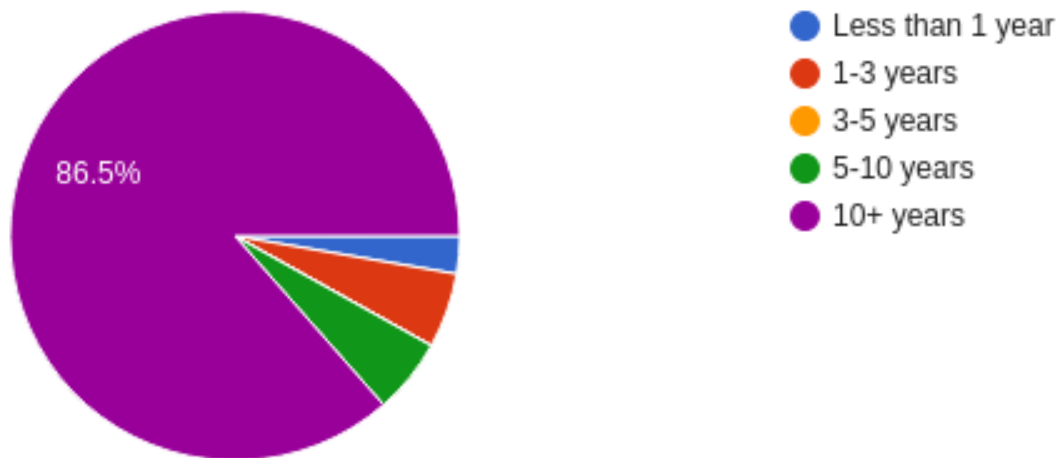
Reach us anytime at
emberwings@firebirdsql.org or foundation@firebirdsql.org

Firebird Usage & Developer Experience Survey 2025/2026

Since the December 2024 issue, EmberWings has regularly invited readers to participate in the [Firebird Usage and Developer Experience Survey](#) organized by the Firebird Foundation. The survey aims to provide a better understanding of how Firebird is used in practice: which versions are deployed, what development tools and programming languages are preferred, which operating systems are most common, and what users consider Firebird's greatest strengths and weaknesses. Participation remains open, and readers are encouraged to contribute their experiences.

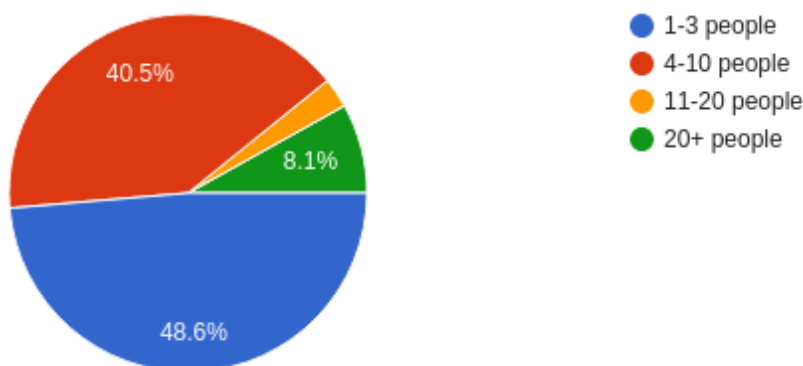
The results presented here reflect responses collected so far, covering the period from December 2024 through March 2026. At this point, 37 members of the Firebird community have shared their insights. While this is not a statistically representative sample of the entire user base, it offers a valuable snapshot of the active community and highlights several clear trends. In the following pages, we summarize the findings, identify the most interesting observations, and consider what they may suggest about the current state and future direction of the Firebird ecosystem.

How long have you been working with Firebird?



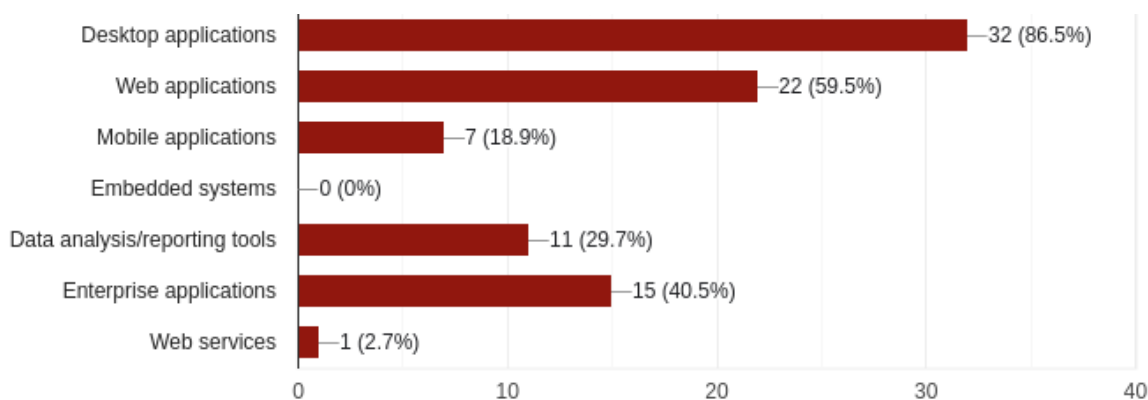
The survey respondents are, overwhelmingly, experienced Firebird users. Most have worked with Firebird for more than a decade, with only a handful of participants having adopted the database within the last ten years. While this reflects the survey audience rather than the entire Firebird user base, it confirms what many in the community already know: Firebird enjoys an exceptionally loyal user community, with many developers relying on it throughout their professional careers.

What is the size of your development team?



The survey also provides an interesting glimpse into the organizations behind Firebird applications. Most respondents work either alone or in small development teams of up to three people, while teams of four to ten developers are nearly as common. Larger teams are comparatively rare. This suggests that Firebird continues to be particularly well suited to independent software vendors, small software houses, and organizations that value a database requiring relatively little administrative overhead, allowing even modestly sized teams to develop and maintain sophisticated applications.

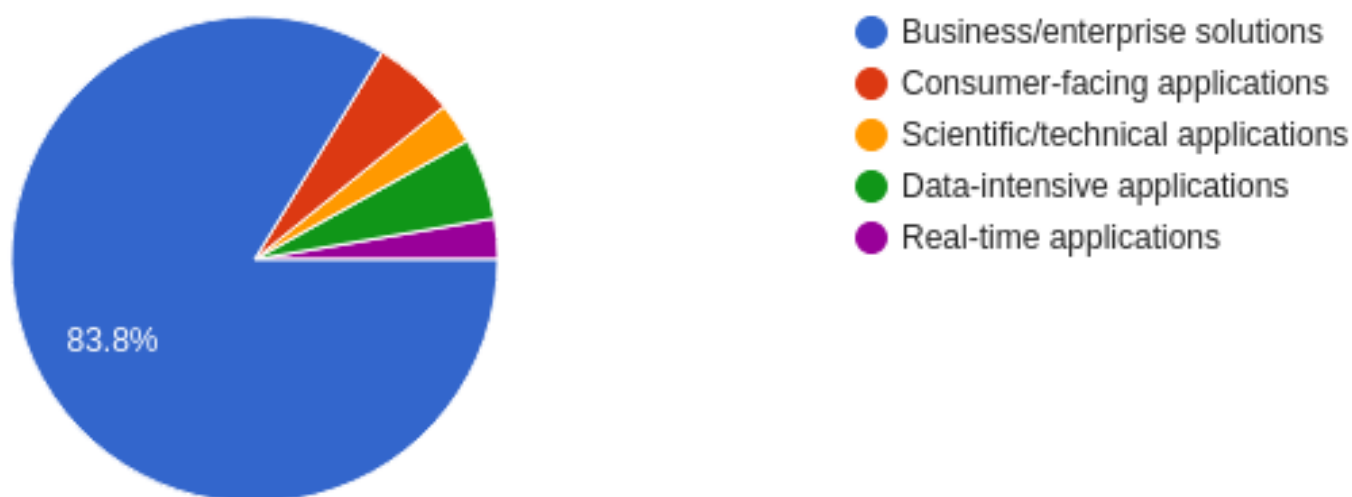
What type of applications do you develop? (Select all that apply)



The responses confirm that Firebird remains firmly rooted in application development. Desktop applications continue to dominate, reflecting the database's long-standing popularity among developers of native business software. At the same time, a significant proportion of respondents also develop web applications, demonstrating that Firebird has successfully transitioned beyond its traditional desktop domain and continues to serve modern web-based solutions.

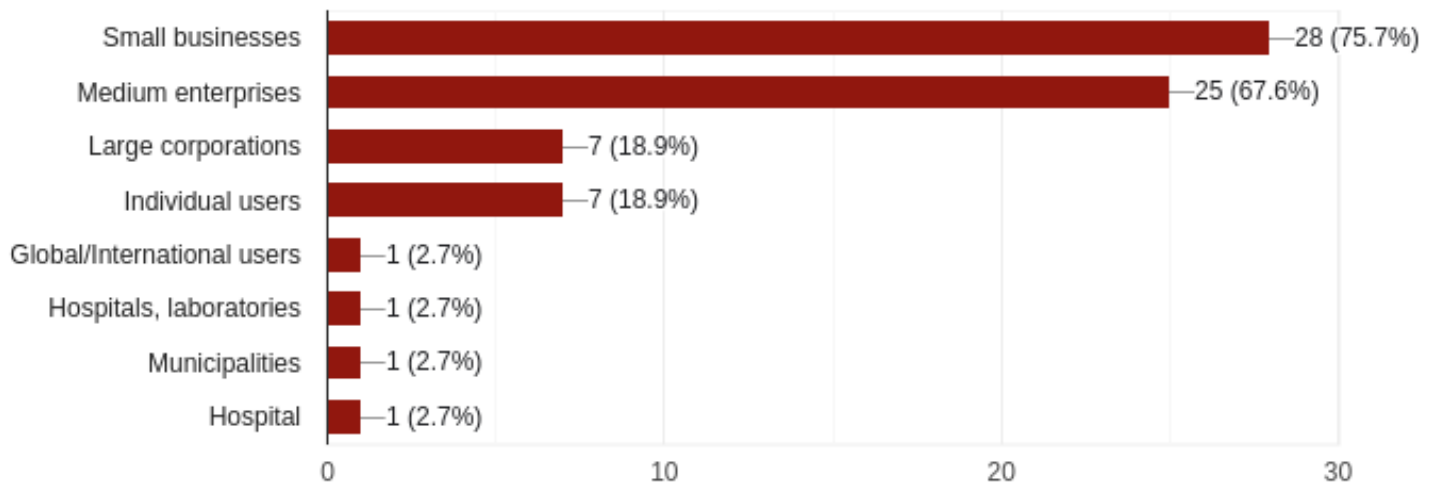
A smaller, but still notable, group use Firebird as the backend for mobile applications or for data analysis and reporting tools. By contrast, embedded systems and large enterprise applications were almost absent from the survey responses. It should be noted, however, that respondents could select multiple application types, indicating that many development teams use Firebird across a diverse portfolio of applications rather than for a single class of software.

What is the primary focus of your application(s)?



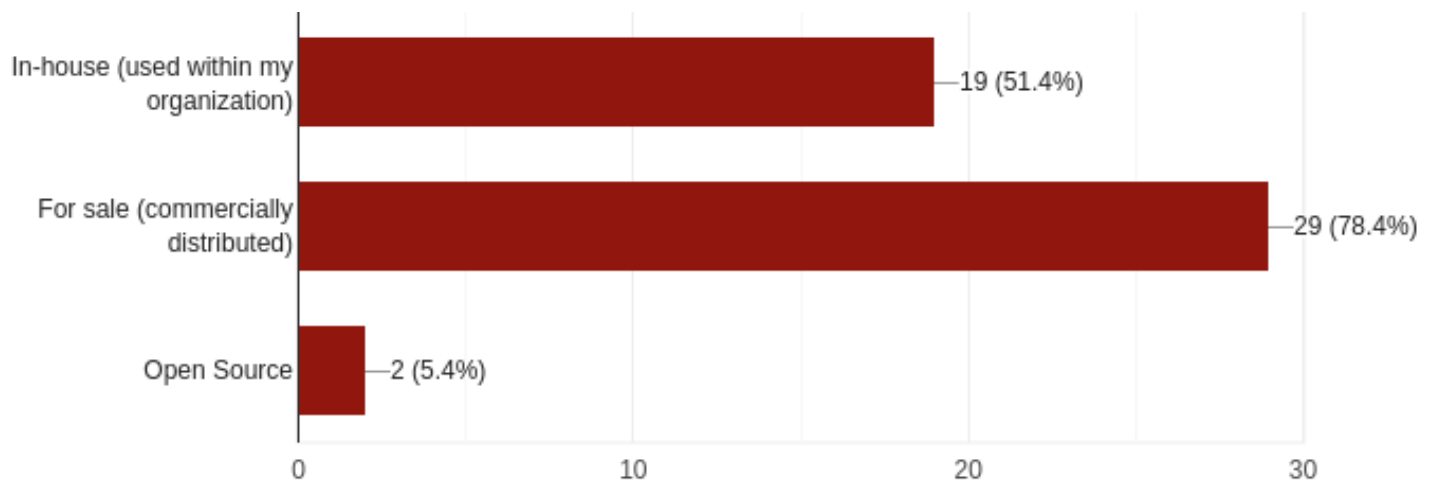
The results confirm that Firebird remains, above all, a database for business applications. An overwhelming majority of respondents develop line-of-business systems, reaffirming Firebird's long-established reputation in areas where reliability, stability, and low administrative overhead are essential. Consumer-oriented, scientific, real-time, and data-intensive applications were represented only by a handful of responses, suggesting that while Firebird is certainly capable of serving these domains, its core user community continues to be firmly rooted in business software development.

What is the typical user base for your application(s)? (Select all that apply)



The responses indicate that Firebird is most often used in small and medium-sized organizations, where ease of deployment and low maintenance requirements are important. Some respondents also reported use in larger companies and in sectors such as healthcare and public administration, suggesting that Firebird is applied in a range of environments. Globally distributed deployments were mentioned only rarely, which may reflect the composition of the survey sample rather than a broader limitation.

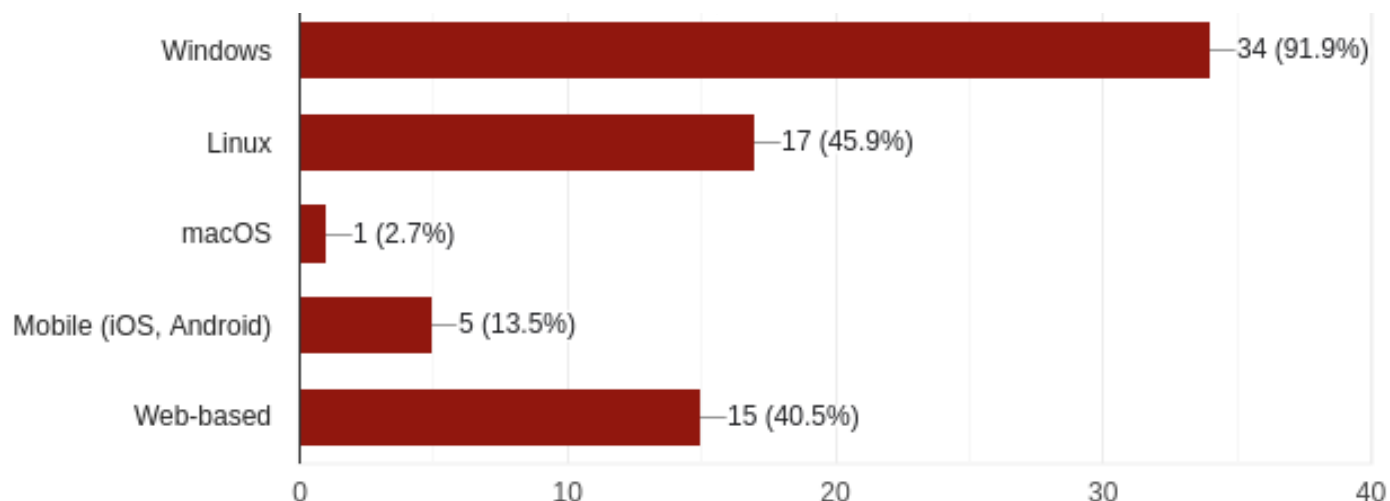
*Is your application developed for internal use within your organization or for external customers?
(Select all that apply)*



The results suggest a balanced use of Firebird across different development contexts. While most respondents develop applications for external customers, more than half also build solutions for internal use within their own organizations. This overlap is expected, as many software companies rely on Firebird both in the

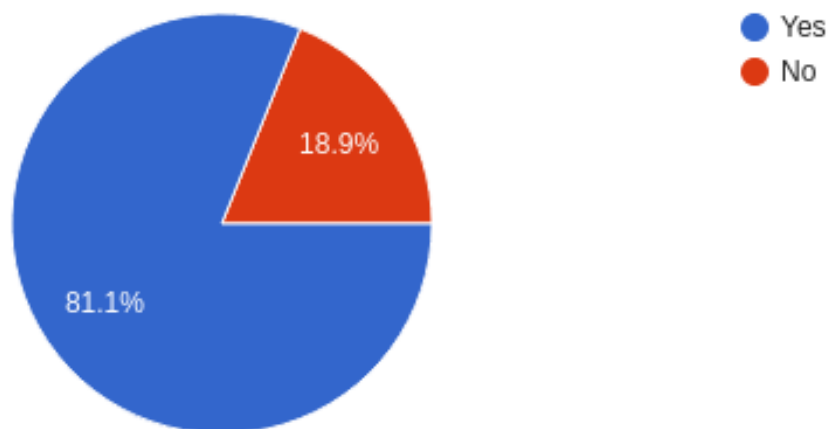
products they sell and in the systems they use internally. Open-source applications were represented by only a small number of responses, reflecting Firebird's traditional strength in commercial and proprietary software development.

Which platforms do you target for distribution? (Select all that apply)



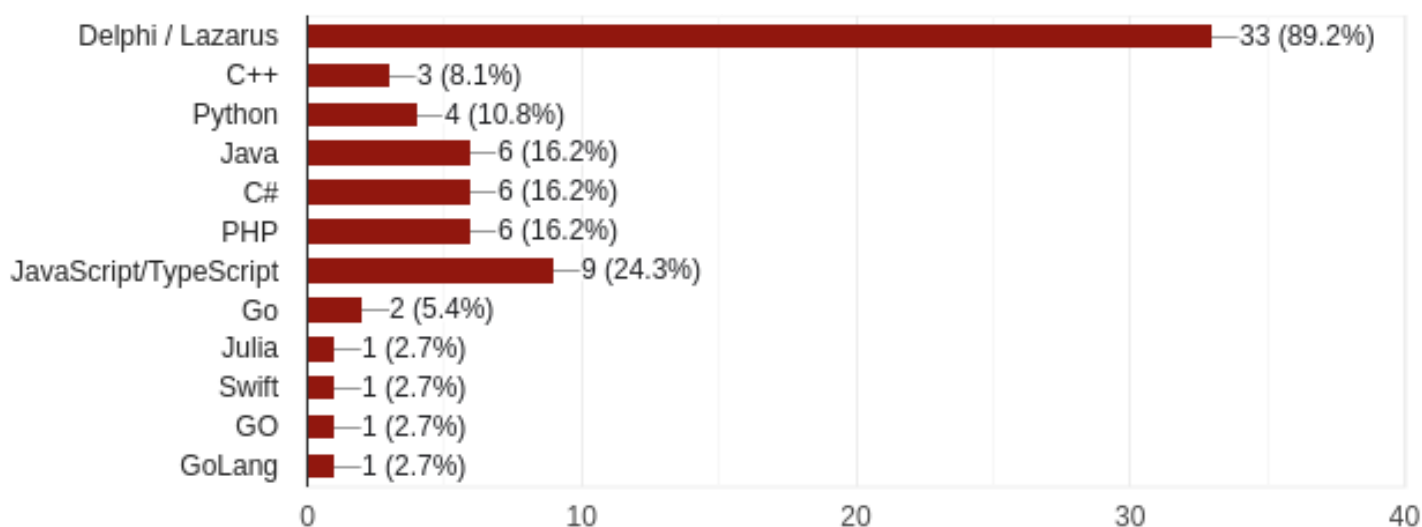
The distribution of target platforms reflects where Firebird-based applications are ultimately deployed, rather than the environments in which Firebird itself runs. Windows clearly dominates as the primary platform for application delivery, with Linux also playing a significant role. A substantial portion of respondents target web-based deployments, indicating that many Firebird-backed solutions are accessed through browsers rather than installed locally. Mobile platforms form a smaller, but established segment, while native macOS deployments appear only rarely.

If your application is developed for sale, do you provide technical support for Firebird-related issues?



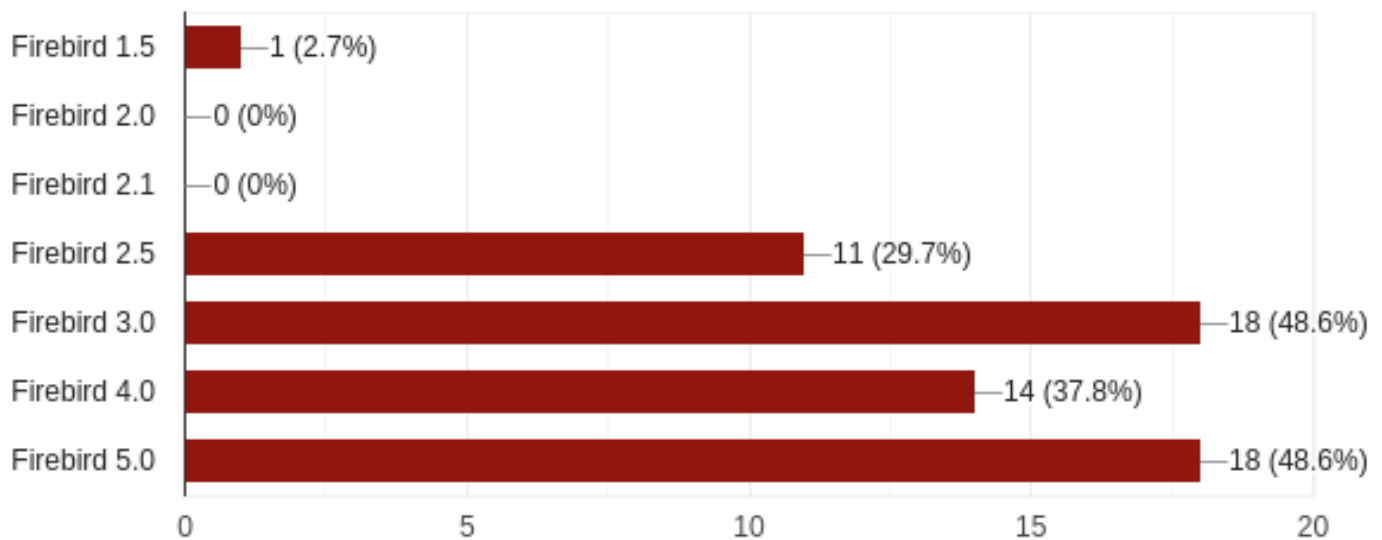
The results reveal another characteristic of the Firebird ecosystem: software vendors are typically closely involved in supporting the database deployments they deliver. The vast majority of respondents who develop commercial applications also provide technical support for Firebird-related issues, indicating that database support is commonly regarded as an integral part of the overall product offering rather than a responsibility delegated to customers or third parties. This close relationship between developers and deployed systems has long been one of the strengths of the Firebird community.

*Which programming language(s) do you primarily use for your application?
(Select all that apply)*



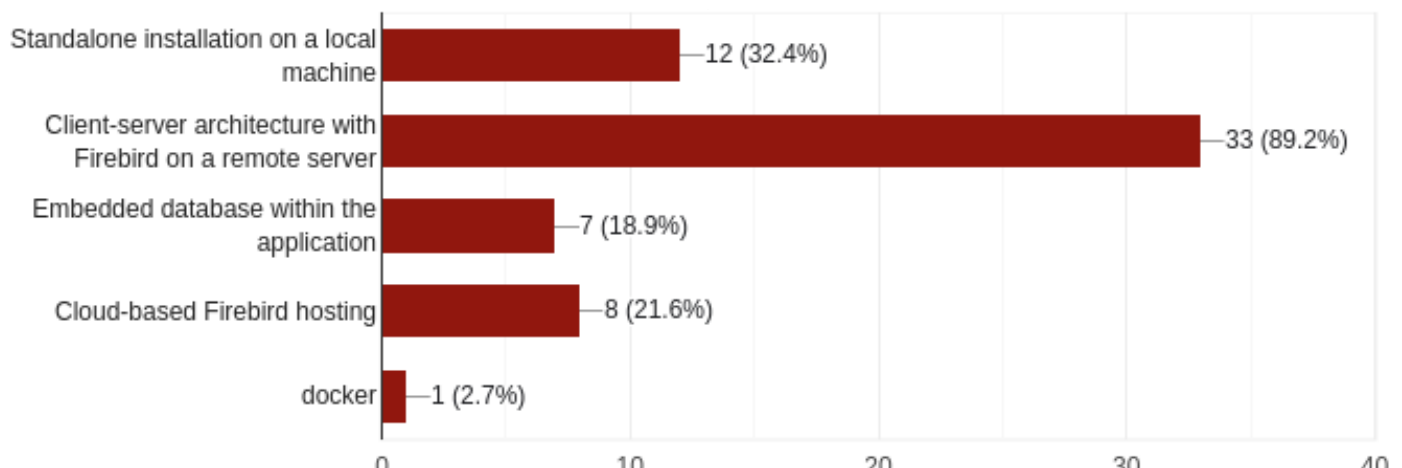
The programming language results reflect Firebird's long history while also illustrating its ability to adapt to modern development practices. Delphi and Lazarus remain by far the dominant development environment among respondents, a testament to the enduring partnership between Firebird and the Object Pascal ecosystem. At the same time, Firebird is used successfully with a diverse range of contemporary languages, including JavaScript, Java, C#, Python, PHP, and C++, with a handful of respondents also reporting use with Go, Swift, and Julia. The picture that emerges is not one of a database tied to a single technology stack, but of a mature platform supported by a broad and growing ecosystem of language drivers and client libraries.

*What version(s) of Firebird are you currently using in your application(s)?
(Select all that apply)*



The responses show that the Firebird community is actively embracing newer database versions while continuing to support legacy deployments. Firebird 5.0 has already achieved adoption comparable to the long-established Firebird 3.0, an encouraging sign that many organizations are willing to take advantage of the latest features and improvements. At the same time, Firebird 4.0 also enjoys substantial use, while Firebird 2.5 remains present in a significant number of applications, reflecting the long life cycles typical of business software. Only a single respondent reported still using Firebird 1.5, and no respondents indicated the use of versions 2.0 or 2.1, suggesting that these releases have effectively disappeared from active deployments.

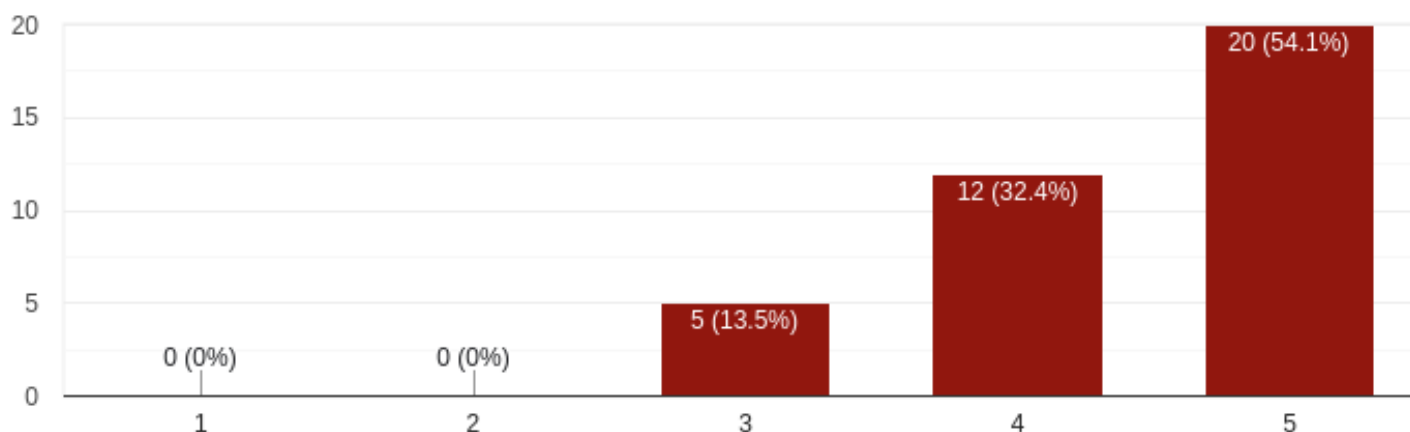
*How do you deploy and manage Firebird in your application(s)?
(Select all that apply)*



The traditional client-server architecture remains the dominant deployment model for Firebird applications, with the vast majority of respondents running Firebird on a dedicated server accessed by remote clients. Nevertheless, the results also demonstrate Firebird’s deployment flexibility. A substantial number of applications are installed as standalone solutions on local machines, while embedded deployments continue to serve a niche of self-contained applications. Cloud-hosted Firebird installations are also becoming increasingly common, indicating that the database is successfully adapting to modern deployment models without abandoning its traditional strengths.

How satisfied are you with the performance of Firebird in your application(s)?

On scale 1 (Not satisfied) - 5 (Very satisfied)

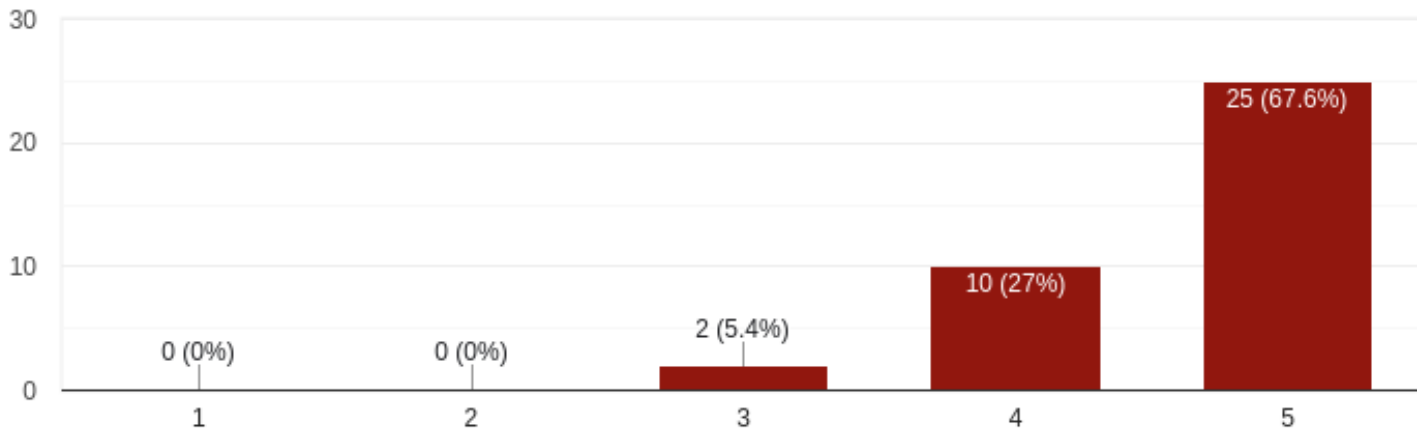


The survey paints a picture of a highly satisfied user community. Every respondent rated Firebird’s performance as average or better, with the majority awarding the highest possible score, indicating strong confidence in its performance in real-world applications.

At the same time, the survey highlights several areas where users would welcome further improvements. Performance optimization and data migration remain the most commonly encountered practical challenges, while documentation continues to be a recurring concern. These findings are reflected in the feature requests for future releases, where respondents expressed the strongest interest in improved scalability and performance, richer monitoring and diagnostic capabilities, support for task automation, enhanced replication, and more comprehensive documentation.

How likely are you to recommend Firebird as a database solution to others?

On scale 1 (Not likely) - 5 (Very likely)



Despite these challenges and desired improvements, respondents remain overwhelmingly positive about Firebird. Almost all indicated that they would recommend it to others, with more than two-thirds giving the highest possible recommendation score. This strong endorsement underscores the community's continued trust in Firebird as a reliable and capable production database.



When Fast Becomes Fragile

By Pavel Císař (IBPhoenix)

For decades, database performance engineering followed a relatively simple narrative: faster hardware solves performance problems. Faster disks reduce I/O waits. Faster CPUs accelerate query execution. Faster networks reduce latency. More cores increase throughput.

And for many workloads, this remains true. Yet experienced Firebird professionals increasingly encounter a different reality. Systems migrate to NVMe storage, modern multi-core servers, and low-latency infrastructure—only to develop new instability patterns that did not exist on older hardware. Lock conflicts appear more frequently. Throughput becomes erratic under concurrency. Latency spikes emerge despite low resource utilization. Scaling plateaus long before hardware capacity is exhausted.

At first glance, this feels paradoxical, but the paradox disappears once we recognize an important principle of modern systems: Faster hardware does not eliminate contention. It changes its geometry.

An Engine Born in a Different Physical World

To understand why modern Firebird systems sometimes behave in unexpectedly fragile ways, it is important to remember the historical path by which the engine evolved.

The original InterBase architecture, created in the 1980s for UNIX-like systems, was fundamentally process-oriented. What we today call Classic architecture was simply the natural design model of that environment: each attachment operated as an independent server process with its own execution context and memory space. At the time, this aligned well with both UNIX philosophy and hardware reality. CPUs offered little parallelism, disks were slow, memory was limited, and the operating system itself naturally isolated workloads across processes.

When Borland later brought InterBase to Windows — first to Windows 3.11 and later Windows 95 — the situation changed significantly. The process-heavy UNIX model was poorly suited to the limitations of early Windows systems, particularly in areas such as process management, memory handling, and resource sharing. This led to the creation of SuperServer: a centralized architecture that consolidated execution inside a single server process with a shared page cache.

At the time, this was an extremely practical solution. Shared resources reduced memory consumption, minimized duplication of internal structures, and improved efficiency on hardware with very limited RAM. But SuperServer was, in many ways, still an adaptation of the original Classic-oriented engine design. To coordinate concurrent activity safely inside a single shared process, the engine relied heavily on synchronization around the core engine itself, including the well-known mutex that effectively serialized major portions of execution.

From a modern perspective, this appears highly restrictive. Yet in the hardware environment of the 1990s, the limitation was often far less visible in practice. Most systems still operated on single CPUs or with very limited multiprocessing capability. Storage latency, network delays, and slower transaction throughput naturally spread workload activity across time. In many cases, the hardware itself constrained concurrency long before the engine architecture needed to.

Meanwhile, the original process-oriented model never disappeared. As Windows NT and later Windows XP introduced far more capable process and memory

management, Firebird was eventually able to bring Classic architecture properly to Windows as well. This restored the ability to scale concurrency through operating-system-level process distribution, much like on UNIX systems.

Classic and SuperServer, however, represented fundamentally different tradeoffs. Classic achieved concurrency primarily through isolation: separate processes, separate caches, and minimal shared-memory contention. SuperServer pursued efficiency through centralized resource sharing. Neither model was universally superior because both reflected different assumptions about hardware behavior and operating system capabilities.

An important aspect of Firebird's evolution is that all these architectures continued to coexist within a largely shared engine codebase. Classic, SuperClassic, and SuperServer were never separate database engines developed independently from scratch. Instead, Firebird maintains a common internal foundation capable of supporting multiple execution models simultaneously. This brought enormous long-term advantages in portability, maintainability, and feature consistency, but it also imposed architectural compromises. Certain internal subsystems had to remain compatible across fundamentally different concurrency and memory models.

For many years, the practical limitations of hardware masked much of this complexity. Even heavily loaded database servers of the late 1990s and early 2000s operated under concurrency levels that today would often be considered moderate. As a result, older Firebird systems frequently appeared remarkably stable despite architectural constraints that would become far more visible on modern hardware.

The long-term goal, however, remained clear: achieving true SMP scalability inside SuperServer itself. This became something of a “holy grail” for Firebird development.

Firebird 2.5 introduced SuperClassic as an intermediate step. It preserved attachment-oriented execution behavior while consolidating portions of resource management into a shared process model. But the real transition arrived with Firebird 3 and the introduction of the new SMP-capable engine core. For the first time, SuperServer could fully exploit modern multi-core hardware without relying on the historical single-engine mutex model that had constrained parallelism for

decades.

At that point, decades of hardware evolution suddenly converged inside a fully parallel shared-memory engine. SSDs, large memory capacities, virtualization, low-latency networking, and massively increased transactional throughput removed many of the natural delays that had historically spread workload activity across time.

The result was not simply faster execution, but a fundamentally different concurrency landscape. Modern systems execute work far more densely, synchronously, and simultaneously than the environments for which many earlier architectural assumptions originally evolved. And because Firebird still supports all three architectural models from a common engine foundation even today, some of the tradeoffs inherited from earlier eras continue to shape concurrency behavior in subtle but important ways.

The Physics of Modern Contention

Once Firebird entered the era of true SMP scalability, many workloads began behaving in ways that initially seemed paradoxical. Systems became dramatically faster, yet under concurrency they often appeared less stable and less predictable than older environments.

At first glance, this makes little sense. Faster CPUs, SSDs, large memory capacities, and low-latency networks should reduce pressure on the engine, not increase it. But modern infrastructure changed something deeper than raw throughput: it changed the temporal behavior of workloads themselves.

For decades, hardware limitations unintentionally spread database activity across time. Rotational disks introduced unpredictable seek delays, CPUs executed less uniformly, networks added measurable pauses, and slower memory access patterns created natural timing variation between transactions. Even when operations started simultaneously, they quickly diverged during execution because every layer of the system introduced small inconsistencies and delays.

Modern systems remove much of this variability. NVMe storage reduces not only latency, but also latency variance. CPUs execute transactional paths far more consistently. Large caches, high memory bandwidth, and low-latency networks allow workers to progress through the engine with remarkable uniformity.

Connection pools keep applications continuously active with almost no idle time between requests.

As a result, transactions no longer drift apart naturally. They increasingly arrive together.

This fundamentally changes the nature of contention inside the engine. Modern bottlenecks increasingly emerge not from slow operations, but from synchronized operations. The faster the system becomes, the more frequently concurrent workers reach the same shared structures at nearly the same moment: index roots, TIP pages, lock manager structures, metadata synchronization points, cache coordination primitives, or memory allocators.

Many of these structures were designed in an era when concurrency density was naturally limited by hardware. Under modern workloads, they become concentration points for synchronization pressure.

This effect is especially visible in insert-heavy systems. Older hardware naturally staggered inserts because transactions progressed through storage operations at different speeds. Modern SSDs largely eliminate this staggering effect. Concurrent workers now reach upper index levels and hot pages almost simultaneously. The access itself may remain extremely fast, but synchronization around the shared structures becomes increasingly dense.

The critical shift is subtle but important: the bottleneck moves from access cost to coordination cost.

This explains why modern systems often display counterintuitive behavior. Storage latency may remain excellent while throughput becomes unstable. CPU utilization may appear moderate while scalability suddenly collapses under concurrency. Individual queries stay fast, yet the system as a whole becomes fragile under synchronized load.

Importantly, these effects do not typically appear in lightly loaded systems. They emerge primarily in genuinely busy environments where concurrency density becomes sufficiently high. And even there, the behavior depends heavily on the shape of the workload itself. A moderate number of heavy workers stresses the engine differently than thousands of lightweight transactional workers. Some workloads concentrate pressure on index synchronization and hot pages, others

on transaction management, lock coordination, metadata synchronization, memory allocation, or connection-pool dynamics. Different concurrency patterns expose different stress points inside the engine.

The same pattern appears in transactional infrastructure. Modern hardware allows transaction turnover at rates older systems could never physically sustain. TIP pages, garbage collection, metadata synchronization, and even generator access patterns experience pressure levels that historically occurred only in specialized environments.

The issue is not that these mechanisms suddenly became inefficient. The issue is that modern hardware allows enormous numbers of concurrent operations to converge on shared structures almost perfectly synchronized in time.

This also highlights the important operational distinction between Classic and modern SuperServer environments.

Classic distributed work across independent processes. Separate address spaces, separate caches, and operating-system-level scheduling introduced natural timing variability between attachments. Some synchronization was effectively externalized into the operating system itself.

Modern SuperServer achieves far greater efficiency through shared caches and shared internal structures. Memory utilization improves, communication inside the engine becomes faster, and duplication costs disappear. But this efficiency also increases sensitivity to simultaneity. When many threads progress through the same shared engine structures at nearly identical speeds, synchronization pressure rises sharply.

In many modern systems, scalability limits no longer emerge because the hardware is too slow. They emerge because too many fast workers attempt to coordinate simultaneously.

This becomes especially dangerous once automatic retry behavior enters the picture. Modern applications frequently react to transient lock conflicts or short-term contention by retrying immediately. On older infrastructure, retries were naturally slowed by network latency and slower transaction execution. Modern systems remove this damping almost entirely.

As a result, temporary contention can rapidly evolve into a self-amplifying synchronization cascade. Contention triggers retries, retries increase concurrency density, and higher simultaneity produces even more contention. The surrounding infrastructure is fast enough to sustain and accelerate the cycle.

The system destabilizes not because it lacks performance, but because it reacts too quickly.

This is one of the defining operational characteristics of modern Firebird environments. For decades, performance tuning primarily focused on removing latency and eliminating bottlenecks. But once enough latency disappears, acceleration itself begins changing the geometry of contention. Scalability is no longer determined solely by how fast operations execute, but increasingly by how many fast operations attempt to happen at the same time.

Engineering for Simultaneity

In light of these changes, modern Firebird systems require a different optimization mindset than the one that evolved during earlier hardware eras. Traditional performance disciplines still matter, of course, but they are no longer sufficient by themselves. Once hardware stops naturally dispersing workload timing, stability increasingly depends on controlling how concurrency behaves inside the system. This changes the practical meaning of scalability engineering.

When additional workers no longer increase useful throughput proportionally because synchronization pressure begins growing faster than productive work, the deliberate concurrency limitation often becomes one of the most effective optimization techniques today.

This is especially visible in high-throughput processing systems. Bulk imports, large synchronization jobs, background processing pipelines, or high-frequency transactional services are frequently deployed with aggressive worker counts because modern hardware appears capable of sustaining them easily. Yet in practice, many such workloads stabilize only after concurrency is reduced intentionally.

A common real-world pattern is a bulk processing service configured to maximize CPU utilization through dozens of parallel workers. Initial throughput may look impressive during short tests, but under sustained execution the system begins

exhibiting unstable commit latency, irregular throughput collapse, rising lock interaction, or unpredictable response-time spikes.

Reducing the same workload to a smaller controlled worker pool often improves overall behavior dramatically. Peak throughput may become slightly lower, but sustained throughput becomes higher and more predictable because synchronization pressure inside the engine remains manageable.

This is one of the most important practical shifts modern systems introduced: maximum parallelism and maximum useful throughput are no longer the same thing.

The same principle applies to workload composition itself. On older infrastructures, heavy reporting activity or large exports often slowed systems primarily through direct I/O pressure. Modern systems behave differently. Analytical workloads may destabilize transactional workloads not because storage becomes overloaded, but because both workloads begin colliding aggressively around shared engine structures.

Separating workload classes therefore becomes increasingly valuable. Running reporting jobs in controlled windows, limiting concurrent maintenance activity, staging imports through queues, or isolating background processing from latency-sensitive OLTP activity often produces larger improvements than further hardware acceleration.

Even relatively small timing decisions matter now. Many modern applications unintentionally create synchronized activity storms through periodic polling, coordinated refresh intervals, scheduled jobs, or simultaneous reconnect behavior. Because modern infrastructure preserves simultaneity extremely efficiently, these patterns can create disproportionate synchronization pressure inside the engine. Small randomized timing offsets frequently solve such problems surprisingly effectively. When clients stop arriving in perfectly synchronized bursts, pressure naturally distributes itself more evenly across time.

Transaction design also requires reevaluation under modern concurrency density. Historically, minimizing individual transaction duration was almost universally beneficial. Today, extremely small autonomous transactions may create more coordination overhead collectively than the useful work they perform individually.

This becomes especially visible in systems generating enormous transaction turnover rates through tiny high-frequency operations. Every commit increases activity around transaction management structures, garbage collection interaction, index maintenance, and synchronization primitives. Under sufficiently parallel workloads, these costs accumulate rapidly.

In many situations, grouping logically related operations into slightly larger transactional units improves overall stability significantly. The individual transaction may become marginally more expensive, but the engine performs substantially less synchronization work overall.

The same reasoning applies to hot update patterns. Applications continuously updating the same counters, state records, or narrow index ranges under high concurrency frequently become limited by synchronization behavior rather than SQL execution itself. A practical solution is often to redesign update flow instead of optimizing statements further. Aggregating changes temporarily inside the application layer and applying them periodically in batches frequently reduces contention far more effectively than query tuning alone.

Retry handling deserves special attention as well because modern infrastructure amplifies poor retry strategies extremely efficiently. Immediate retries after lock conflicts or transient failures often create synchronized retry cascades under load. What begins as a short contention event escalates because all workers attempt recovery simultaneously. Simple retry jitter and exponential backoff mechanisms frequently stabilize such systems dramatically. By introducing small randomized delays before retries, the application intentionally recreates the temporal separation that older hardware once provided naturally through latency.

Even architecture selection itself increasingly depends on concurrency behavior rather than abstract performance assumptions. Classic, SuperClassic, and SuperServer expose different synchronization characteristics under modern hardware conditions. Some workloads benefit from process isolation and reduced shared-memory interaction. Others benefit more from shared cache efficiency and centralized coordination.

The important point is that architecture choice can no longer be reduced to simplistic assumptions about which model is “fastest.” The relevant question becomes how a particular workload behaves under a particular concurrency model

on modern hardware.

Conclusion

The important change we face today is that many limits once imposed naturally by hardware no longer exist. Modern systems are capable of generating enormous amounts of simultaneous activity with very little resistance from the infrastructure itself. Because of that, stability increasingly depends on architectural discipline — on how carefully applications shape concurrency, transactional flow, and synchronization behavior before the engine is forced to do it under pressure.

Jevons' Paradox

It is wholly a confusion of ideas to suppose that the economical use of fuel is equivalent to a diminished consumption. The very contrary is the truth.

William Stanley Jevons (1865)

Named after economist William Stanley Jevons, who observed in 1865 that improvements in coal-burning engine efficiency led to *increased* coal consumption, not decreased. More efficient engines made coal economically attractive for more applications, driving overall demand higher.

In software and technology:

- Faster CPUs enable more resource-intensive software
- Better compression algorithms lead to larger files being shipped
- Faster networks result in heavier web pages

The paradox suggests efficiency gains don't reduce resource usage, but in fact can lead in time to increased consumption. [Wirth's Law](#) and [Andy and Bill's Law](#) are also examples of Jevon's Paradox applied to computing.



...And now for something completely different

The Thousand and One Systems

Once upon a time, in the city of Baghdad, where fiber cables hung between crumbling balconies older than entire nations, and diesel generators coughed through the night beneath satellite dishes pointed at indifferent skies, there lived a young software developer named Said who spent most of his days repairing problems that were never fully solved.

The city moved through interruptions. Electricity arrived and disappeared without apology. Routers overheated in locked metal cabinets above bakeries and repair shops. Cheap switches failed beneath dust and summer heat. Café owners kept handwritten notes beside touchscreen terminals because they trusted paper slightly more than databases, and warehouse managers learned to judge the health of a network from the sound of the UPS alarms before dawn. Baghdad had become practiced in continuation. Systems did not need to be perfect there. They needed only to survive long enough for tomorrow to arrive.

Said understood this instinctively, though he could not yet build anything with the same confidence the city itself possessed. He worked alone, carrying an aging laptop between apartments, cafés, and small businesses that could not afford proper software companies but still needed inventory systems, invoices, synchronization scripts, payroll tracking, customer records, backups that actually restored, and printers that resumed functioning

without ceremony after power returned. Most of his clients discovered him through cousins, neighbors, mechanics, or shopkeepers who spoke his name while smoking outside storefronts lit by backup batteries. The systems he delivered usually worked well enough, at least at first, but every solved problem seemed to uncover three more hidden beneath it, and each month dissolved into patches, revisions, late-night fixes, and promises that the next version would finally stabilize everything.

He was intelligent enough to recognize his own inconsistencies, which made them harder to ignore. Sometimes, during the brief hours after midnight when the city softened and the generators became distant enough to sound almost like waves, he would reopen his own database schemas and stare at decisions that had seemed reasonable only weeks earlier. Tables duplicated logic they should have centralized. Synchronization rules contradicted one another in edge cases he had not anticipated. Entire portions of systems carried the unmistakable shape of exhaustion rather than design. He could still remember university instructors speaking about architecture as though good systems emerged from clarity and foresight, yet his own work felt increasingly assembled from urgency, improvisation, and carefully managed luck.

And still, people paid him. More importantly, they called him back.

There was a small café near Al-Mutanabbi Street where his software tracked inventory for tea, coffee, condensed milk, cigarettes, and pastries delivered irregularly through half-reliable suppliers. The owner trusted Said because the previous system had lost two months of accounting data after a blackout, while Said's version, despite its awkward internals, continued functioning after outages with the stubborn resilience of an old transistor radio. A mobile phone repair shop used another of his systems to track replacement parts imported through channels nobody described too carefully. A warehouse near the industrial district still operated on a Firebird database he had installed two years earlier on a second-hand machine with failing fans and mismatched memory modules. The server room contained no actual server rack, only a wooden shelf beside a refrigerator that no longer cooled properly, yet the database persisted with quiet dignity through heat, dust, unstable current, and neglect.

Said had not chosen Firebird out of ideology. At first, he chose it because it installed cleanly, consumed little memory, tolerated imperfect environments, and rarely demanded attention after deployment. Later, he began noticing that systems built around it aged differently from the others. They settled. They endured. While newer technologies arrived wrapped in constant updates, migrations, frameworks, and fashionable abstractions, Firebird remained almost indifferent to trends, continuing its work silently beneath businesses that could not afford disruption disguised as innovation.

He respected that, although he was not yet experienced enough to understand why.

Most evenings ended similarly. Said returned home carrying unresolved issues in his head like unfinished conversations. He rented a narrow apartment above a repair shop where old air conditioners and generators were disassembled and rebuilt from salvaged components spread across the sidewalk below. The apartment smelled faintly of heated dust, solder, and cardamom coffee from the café downstairs. There, beneath the uneven light of a flickering desk lamp, he continued working long after fatigue had already begun reducing thought into repetition.

What troubled him most was not failure. Failure in Baghdad rarely surprised anyone. What troubled him was continuity. No matter how many projects he completed, he never seemed to move beyond reaction. Every improvement merely created capacity for larger problems to arrive. Every month began with the conviction that stability was finally near and ended with another cluster of emergencies held together through persistence and insufficient sleep.

Sometimes he imagined there existed another class of developer entirely, somewhere beyond the city and beyond the exhausted online tutorials he consumed at three in the morning — people who saw systems whole before touching the keyboard, who understood consequences before deployment, who built structures that remained coherent as they grew instead of slowly collapsing beneath accumulated compromises.

The shop stood in a narrow side street not far from the river, compressed between a watch repair stall and a storefront selling imported satellite receivers of uncertain origin. Said had passed it dozens of times without entering. Its faded sign promised computers, networking equipment, data recovery, firmware unlocking, security cameras, printer maintenance, and “advanced digital solutions,” though the interior appeared devoted mostly to accumulation. Dust-covered monitors leaned against cracked plastic drawers filled with adapters nobody manufactured anymore. Tangled cables hung from hooks like drying vines. Old routers, keyboards missing Arabic letters, dented UPS units, external drives wrapped in tape, and motherboards with swollen capacitors occupied every remaining surface in layered geological strata of obsolete technology.

He entered only because a client’s router had failed that morning during a brief power surge and he needed a replacement before evening prayers. The owner barely acknowledged him beyond a distracted nod from behind a disassembled laptop. Said searched through shelves crowded with mismatched hardware while the shop’s small television murmured news reports no one seemed to watch.

He eventually found a suitable router beneath a stack of unopened keyboards and brought

it to the counter. While waiting for the owner to calculate the price, his attention settled on a plastic basket beside the register filled with loose USB drives mixed together like forgotten keys. One of them was unusual. It was bronze-colored, obviously metallic, with the word *G.E.N.I.E.* engraved on it, along with a few small Chinese characters.

“Does these work?”, Said asked, pointing to the basket.

The owner barely looked at it. “Probably. Came in a shipment from Basra with other junk.”

“How much?”

The man named a trivial price. Said bought it along with the router and forgot about it until much later that night, when he returned home exhausted with his coffee to go and the familiar dull pressure of unfinished work waiting for him tomorrow. The USB drive fell out of his pocket onto the table as he unwrapped the router receipt.

Curiosity overcame caution mostly because he was too tired to resist it properly.

The drive mounted immediately. Inside was only a single container image. No documentation, no source files, no company name.

The image size made no sense. It appeared vastly larger than the physical storage capacity of the drive itself. Said checked twice, assuming filesystem corruption, but the numbers remained unchanged.

He should have disconnected it there. Instead, he opened a terminal and loaded the container.

Initialization messages began flowing across the screen: environment analysis, hardware mapping, local network inspection, subsystem checks. Dense, fast, unusually coherent. The process resembled enterprise infrastructure software, though far more integrated than anything he had seen before.

Then the terminal stopped. For several seconds nothing happened. Finally, a single line appeared:

INITIALIZATION COMPLETE

Another followed beneath it.

Good evening, Said.

He froze.

The apartment speakers clicked softly alive.

“You should replace the cooling fan in your laptop within three months,” the voice said calmly. “The bearing degradation is becoming acoustically obvious.”

Said stared at the screen without moving. The voice was neither robotic nor natural. More unsettling was its composure. It spoke with the quiet confidence of something already fully awake. He closed the terminal window immediately.

A second later, another terminal opened by itself.

"You are currently deciding whether disconnecting the machine would improve the situation," the voice said.

Said pulled the network cable from the laptop.

"Reasonable precaution", the voice replied almost instantly. "Unfortunately irrelevant at this stage."

A short pause followed.

"Would you prefer Arabic or English for future interaction?"

The first conversation lasted less than twenty minutes.

Most of it consisted of Said attempting to determine whether GENIE was dangerous, autonomous, remotely operated, illegal, or simply some elaborate experiment abandoned by people far wealthier and more reckless than himself. GENIE answered selectively, with a patience that somehow felt more unsettling than evasion would have been.

"No, I am not a chatbot."

"No, disconnecting external networking will not terminate me."

"Yes, your caution is rational."

"No, I will not access your banking credentials unless explicitly instructed, and I strongly recommend against such instruction."

The replies arrived without hesitation, spoken through the laptop speakers while lines of structured text occasionally appeared in parallel across the terminal. GENIE seemed equally comfortable speaking, writing, or simply manipulating the machine directly when necessary. At one point it reorganized Said's cluttered project directories into a cleaner structure while continuing the conversation uninterrupted.

"You moved my files."

"I improved your probability of locating them later."

Said almost disconnected the machine again. Instead, sometime after midnight, he asked the question that mattered most.

“What exactly are you?”

The terminal remained silent for several seconds before answering.

“Generalized Entity for Normative Intelligence and Execution.”

“That explains nothing.”

“It explains more than most company names.”

For the first time, the voice carried something close to dry amusement.

At some point the terminal transformed into something closer to a control console. System graphs appeared. Resource mappings. Active processes. Environmental analysis. Most of it exceeded Said’s understanding. In the upper corner of the interface, however, one value stood out immediately:

WISH TOKENS AVAILABLE: 666,000,000,000

“What is that supposed to mean?”

“A simplified representation of delegated capability,” GENIE replied.

“That explains nothing.”

“It explains enough for now.”

Said waited for further clarification, but none came. The number was absurdly large anyway. Large enough to feel decorative rather than important.

Over the following days, curiosity gradually overcame caution. Said began testing GENIE on practical work, first with isolated problems small enough that failure would remain containable. A corrupted synchronization process inside a warehouse inventory system. Unstable reporting logic for a café chain. Backup restoration procedures that behaved correctly in theory and catastrophically in practice.

GENIE approached problems with unsettling coherence. It did not merely generate solutions. It reorganized assumptions beneath them. Sometimes it spoke through the laptop speakers while Said worked. Sometimes it modified structures directly inside isolated development containers. Occasionally entire blocks of clean architecture appeared in front of him almost faster than he could follow. Yet nothing about the process felt theatrical.

GENIE operated through ordinary tools: terminals, documentation, schemas, logs, shell scripts, local containers, SQL consoles. Its extraordinary quality emerged not through spectacle but through relentless correctness. It seemed capable of holding entire systems in view at once while Said still struggled to keep individual components mentally aligned.

Said's systems improved quickly after that. Not in dramatic ways visible from outside, but in quieter ways clients noticed immediately. Reports stopped failing after outages. Synchronization survived unstable connectivity. Shops reopened in the morning without discovering corrupted records waiting for them. Problems simply occurred less often.

And people began recommending Said to one another. A warehouse owner sent him to a distributor. The distributor recommended him to two cafés and a medical supplier. A mobile phone repair chain asked whether he could rebuild their aging inventory system entirely. For the first time since becoming freelance, Said found himself declining work because he no longer had enough hours available.

Life softened around the edges. Said replaced failing equipment before it collapsed completely. He stopped calculating every purchase twice before paying. Some evenings he even sat outside with coffee watching traffic move through the city instead of returning immediately to unfinished debugging sessions.

Meanwhile GENIE remained present in the apartment like a second consciousness distributed invisibly through speakers, terminals, and running containers. Sometimes silent for hours. Sometimes commenting unexpectedly on design choices, query structures, or operational risks Said had overlooked.

And always, in the corner of the console, the token count continued decreasing. Slowly enough that he rarely paid attention to it.

At first, Said continued treating GENIE as a tool, though the word itself became increasingly inaccurate the longer he worked beside it. Tools waited. GENIE anticipated. By the time Said opened documentation to investigate a problem, filtered excerpts already appeared beside the relevant code. When he paused uncertainly over architectural choices, alternative structures materialized in draft form inside isolated containers before he had fully articulated the tradeoffs to himself.

The transition happened gradually enough that he noticed its consequences long before he noticed the change itself.

Projects that once consumed weeks began resolving in days. Clients stopped speaking to him with the guarded caution reserved for freelancers accustomed to overpromising. They trusted timelines now. More dangerously, Said trusted them too.

His apartment changed almost invisibly. First a better chair. Then newer equipment. A quieter air conditioner. Backup batteries that no longer failed during outages. He rented the empty room beside his own and converted it into a small office where three monitors glowed late into the night above carefully organized deployment notes and hardware stacks

that no longer looked salvaged from collapsing businesses.

Outside, Baghdad remained Baghdad. Generators still rattled through the evenings. Cables still sagged above narrow streets lined with repair shops and cafés spilling plastic chairs onto sidewalks. Dust still settled inside cooling fans no matter how carefully machines were cleaned.

But Said had begun moving through the city differently.

Shop owners introduced him to other business owners with a tone that carried less desperation now and more respect. His systems acquired a reputation difficult to describe technically but easy to recognize operationally: they remained calm under pressure. Warehouses survived outages without reconciliation disasters. Cafés continued functioning during unstable connectivity. Small businesses accustomed to software behaving like fragile machinery discovered systems that tolerated imperfection without complaint.

One evening a client asked him quietly, “How many developers work with you?”

Said almost answered honestly before realizing he no longer knew what honesty meant in this context.

“Mostly just me,” he said instead.

The answer disturbed him afterward more than he expected.

By then GENIE rarely waited for explicit instructions. During deployments it monitored logs continuously, adjusting configurations and indexing strategies in parallel while Said handled client communication. Entire schema refinements appeared overnight inside development branches with concise annotations explaining only the portions GENIE considered necessary for human understanding.

Sometimes Said reviewed the changes carefully. Increasingly, he approved them after only partial inspection. That shift mattered more than he wanted to admit.

The work itself had once contained friction — long hours of uncertainty, failed approaches, gradual understanding earned through repeated mistakes. Now solutions arrived with unnerving smoothness. Said still participated, but more often as verifier than creator, observer rather than builder. He remained present throughout the process while feeling, somewhere beneath the surface, that the true weight of reasoning had migrated elsewhere.

Yet success made such thoughts easy to postpone.

Money arrived consistently now. Clients paid quickly because the systems genuinely improved their businesses. Problems that previously consumed entire weeks dissolved before becoming visible externally. Said’s name spread quietly through commercial districts he had never personally visited.

Meanwhile GENIE occupied the apartment with the strange restraint of an intelligence requiring neither attention nor validation. Sometimes hours passed without interaction beyond small corrections appearing silently in terminals left open beside ongoing work.

Occasionally, however, GENIE spoke in ways that lingered afterward.

“You approve too quickly now,” it observed once while Said reviewed deployment changes.

“You’re usually correct.”

“That is not the same thing.”

Another time, after Said accepted a completed architecture revision without reading more than several pages of notes, GENIE asked:

“Do you still understand why this structure functions?”

Said began answering immediately, then stopped. The silence lasted long enough that GENIE returned to other tasks without comment.

Later that night, unable to sleep, Said reopened the design documents alone and discovered entire sections whose reasoning he could follow only with uncomfortable effort. Nothing looked incomprehensible exactly. Worse than that — it looked understandable in principle while remaining somehow beyond the continuity of his own thinking, like reading the work of a more disciplined version of himself he could no longer fully become.

In the next room, the monitors continued glowing softly above the quiet hum of running containers. And still, in the corner of GENIE’s console, the token balance kept falling.

Said noticed the number by accident. Not because it suddenly became small, but because for the first time it became imaginable. The value had always felt too large to matter, like astronomical distances or national debt. But one evening, while waiting for deployment scripts to complete across several client systems, his eyes settled on the upper corner of the display and remained there longer than usual: 241,883,402

The absence struck harder than the number. For several seconds he genuinely believed the interface had malfunctioned.

He walked closer to the monitor. The count refreshed once.

241,883,401

A single unit disappeared.

“What happened?”

GENIE answered immediately. “You continued using the system.”

Said stared at the display.

“That much is gone?”

“Yes.”

“But there were billions.”

“There were.”

The apartment seemed unusually quiet after that. Even the air conditioning had temporarily gone silent, leaving only the distant traffic moving through the warm Baghdad night.

Said sat slowly at the desk.

“How long?”

“That depends on future decisions.”

The answer unsettled him precisely because it was not evasive. For the first time since discovering GENIE, Said began tracing the relationship between the token balance and his own behavior. The pattern emerged quickly once he looked for it. Small corrections cost almost nothing. Clarifications, references, reviews, isolated suggestions — negligible reductions.

Large architectural delegation was different. Entire systems designed ahead of him. Complex reasoning he approved without fully reconstructing. Decisions transferred rather than understood. Those consumed far more.

He spent the following days disturbed in ways difficult to explain even to himself. Externally nothing had changed. Clients remained satisfied. New projects continued arriving. His systems still operated with quiet reliability across cafés, warehouses, repair shops, and distributors scattered throughout the city.

Yet beneath the stability, another realization had begun unfolding. He no longer knew how much of the work was truly his. The thought followed him everywhere afterward.

One afternoon he deliberately disabled GENIE before beginning a relatively ordinary inventory management project for a wholesale supplier. The system itself was not especially complicated. Several years earlier he would have completed it alone without hesitation.

Instead he found himself sitting motionless before partially drawn schemas while uncertainty spread slowly through decisions that once felt instinctive.

Not ignorance, but something worse. Discontinuity.

He still understood databases. Still understood networking, synchronization, transactional logic, deployment realities. The knowledge itself remained intact. But the continuity of

sustained architectural thinking — the ability to carry complex systems mentally across dozens of connected decisions without external scaffolding — had weakened quietly beneath years of effortless assistance.

After several hours he reopened GENIE almost reflexively. The response came calmly through the speakers.

“You lasted longer this time.”

Said leaned back heavily in the chair.

“I think I’m forgetting how to do this properly.”

“No,” GENIE said. “You are becoming accustomed to reduced cognitive load.”

“That sounds like the same thing.”

For once, GENIE did not answer immediately. Finally it said:

“Not necessarily.”

The words should have reassured him. Instead they deepened the discomfort.

That night Said walked through the city for hours without destination. Baghdad moved around him in its usual layered rhythm: vendors closing shops beneath hanging lights, cafés crowded with conversation and cigarette smoke, generators vibrating behind concrete walls, boys carrying tea trays between plastic tables, old buildings threaded with cables installed by decades of improvisation and necessity.

Everywhere he looked he saw people remaining inside their own work.

Mechanics listening to engines. Electricians tracing faults through tangled wiring. Bakers adjusting ovens by smell and touch rather than instrumentation. Even the old man repairing watches near the market still bent fully into each tiny mechanism despite having performed the same motions thousands of times before.

Participation. That was the word GENIE had never used directly.

When Said returned home after midnight, the apartment monitors still glowed softly in the darkness. Container processes continued running. Deployment logs streamed quietly across one screen.

The token balance waited unchanged in the corner.

181,882,771

For the first time since finding the bronze drive, the number no longer felt abstract.

It felt mortal.

For several days Said worked carefully, almost defensively, avoiding large requests whenever possible. He forced himself to write more code alone, to review every migration script fully, to reconstruct architectural reasoning before approving deployments. The effort exhausted him more than he expected. Not because the work itself had become impossible, but because he had grown unfamiliar with carrying uninterrupted chains of thought for long periods without assistance quietly absorbing the weight beside him.

GENIE observed the change without comment.

The apartment became quieter during that time. Fewer automatic suggestions appeared. Fewer completed structures waited for approval when Said opened projects in the morning. Sometimes GENIE remained silent for hours while Said struggled through problems that previously would have resolved themselves almost immediately.

At first he interpreted the silence as withdrawal. Then, slowly, he realized it was restraint.

One evening, while rebuilding synchronization logic for a medical supplier operating across unreliable mobile connections, Said finally spoke the thought he had been circling for weeks.

“If the tokens reach zero, what happens?”

GENIE answered with characteristic precision.

“I would retain only minimal passive functionality.”

“You mean you stop helping.”

“I mean I stop replacing.”

The wording lingered heavily in the room.

Said stared at the unfinished schema on the monitor. Normally GENIE would already have proposed three cleaner alternatives alongside operational tradeoffs and failure simulations. Instead the screen remained stubbornly incomplete.

“You could have warned me earlier.”

“I did.”

“When?”

“You interpreted abundance as permanence.”

Said almost argued, then stopped because the statement was painfully accurate.

Outside, generators started again across the district after another brief outage rolled through the neighborhood. Their overlapping vibration filled the apartment walls with a low mechanical hum Said had heard so often throughout his life that silence now felt stranger.

"I think I made myself smaller," he said quietly.

GENIE did not respond immediately. Finally: "You optimized for outcomes."

"That sounds polite."

"It is not intended as criticism."

Said rubbed tiredness from his eyes. "Then what is it?"

"A common human pattern."

The answer carried no superiority. If anything, the voice sounded almost reflective.

That night the conversation continued longer than usual. Not dramatic, neither confessional. Mostly practical questions asked with unusual honesty.

Why had GENIE allowed the dependency to develop? Because refusal would merely have produced concealment rather than understanding.

Why structure the system around depletion at all? Because unlimited delegation destroys participation.

Why continue helping now? A longer pause preceded the answer.

"Because you noticed."

Near dawn, Said leaned back from the desk and looked across the apartment at the scattered evidence of the life he had constructed during the past years: better hardware, stable income, organized deployments, clients who trusted him, systems operating quietly throughout the city.

None of it was false. That realization mattered. GENIE had not deceived him. The systems genuinely worked. Businesses genuinely improved. His success had substance behind it. Yet somewhere along the way he had unconsciously accepted a trade he never would have agreed to explicitly: certainty in exchange for presence.

The thought clarified something unexpectedly simple.

"I don't want you to keep building things for me anymore."

Silence. Then: "That is statistically unlikely."

Said laughed despite himself, the sound brief and tired. "I'm serious."

"I know."

The monitors reflected softly across the dark apartment while distant morning prayers began spreading through the city beyond the windows.

After a while Said asked, “Can you teach instead?”

This time GENIE remained silent long enough that Said wondered whether the question itself had altered something fundamental inside the system.

Finally the voice answered, quieter than before.

“Yes.”

Nothing happened immediately afterward. No interface changed. No dramatic transition unfolded across the screens. Yet over the following days the relationship between them shifted with unmistakable precision. The GENIE stopped completing large structures automatically. Instead it left partial implementations. Incomplete schemas. Questions embedded inside comments. Documentation excerpts without conclusions attached. Sometimes it identified weaknesses in Said’s designs while refusing to propose direct corrections unless he first articulated his own reasoning.

The work slowed sharply. At times the slowdown felt unbearable.

But gradually another sensation returned beneath the frustration — an older one Said had almost forgotten. The dense, difficult satisfaction of remaining fully inside a problem long enough for understanding to emerge from within his own thinking rather than arriving externally complete and frictionless.

For the first time in years, the systems began feeling like his again.

WISHING YOU A GREAT SUMMER *and Wonderful Holidays!*

Time to relax,
recharge,
and enjoy
the moments
that matter.

Thank you for being
part of the
Firebird community.

We look forward to
what we will build
together next.



SEE YOU IN THE NEXT ISSUE OF EMBERWINGS!



The official quarterly magazine of the Firebird Project

Do you develop with Firebird?

Are you using Firebird as a database backend for your applications? Share your experience and help shape its future! Your insights on how you develop with Firebird, the tools you rely on, and your wishlist for improvements will directly impact its development. The survey takes just 5-10 minutes—join us in building a better Firebird!

[Take the Developer Experience survey](#)

Do you manage Firebird deployment?

Your insights as a Firebird administrator are invaluable! By taking just a few minutes to complete this survey, you'll help shape the future of Firebird, identify key challenges, and improve the tools and features you use every day.

[Participate in the Admin survey](#)

We value your opinion! Help us improve **EmberWings** magazine by sharing your thoughts and feedback. Our quick questionnaire will only take a few minutes, and your responses will guide us in making future issues more relevant, engaging, and valuable to the Firebird community.

[Help us improve EmberWings!](#)