Texas Linux Fest 2026 Observations

Borislav Sabotinov · November 6-7, 2026 · Austin, TX

Texas Linux Fest is a volunteer-run community conference, and this year it ran two days at the Commons Conference Center in Austin. Day one was mostly hands-on workshops and day two was talks. AI was a hot topic; instead of treating it as standalone, most presented it as one more component of a Linux system: something you package, run in a container, manage with systemd, put boundaries around, and review before deploying. Here are some observations that stood out.

> Thou shalt not make a machine in the likeness of a human mind.
>
> Frank Herbert, *Dune* (the Orange Catholic Bible)

Don't delegate your thinking

Major Hayden, *Thou Shalt Not Make a Machine in the Likeness of a Senior Engineer* (the talk)

My favorite talk of the conference. Major used Dune to frame the fear around AI. In the book, humanity hands its thinking over to machines, then fights a holy war (the Butlerian Jihad) to destroy them. Surrender and revolution look like polar opposites, but both approaches allow fear to dictate the decision. It's OK to choose neither.

We should not delegate our thinking to AI. Rather we should maintain focus on what we can directly control:

- our own judgment
- what we choose to learn
- what we approve and merge
- what we teach others

Two practices stood out and I plan to incorporate them in my workflow:

Plan far more than you implement. His rule of thumb: spend 10x more time and tokens on planning than on implementation. Be very specific and narrow about what is in scope, and just as explicit about what is out of scope. *A vague boundary is an invitation for the agent to wander*.

Have AI draw a diagram of the before and after. Ask it for Unicode art of the current state (e.g., the bug, the existing flow) and the proposed state (e.g., the fix, the new capability, the redesign). In a PR description, review comments, or documentation it makes changes easier to reason about and digest, and it's a quick check that you understand the change yourself. Something like:

```
CURRENT
client ──> api ──> db
                   x timeout (500)

PROPOSED
client ──> api ──> db
           │       x timeout
           └──> retry w/ backoff ──> db
                   x 3rd fail ──> 503 + alert
```

AlmaLinux and Rocky Linux after CentOS

Both came to be because of Red Hat's late-2020 decision to end CentOS Linux in favor of CentOS Stream. Six years later they hold roughly a quarter of enterprise deployments (one 2026 survey puts them at ~12% each). They started similarly, rebuilding RHEL sources, then diverged in 2023 when Red Hat restricted public access to RHEL source code:

- **AlmaLinux** (backed by CloudLinux, governed by the AlmaLinux OS Foundation) switched from 1:1 to ABI compatibility and builds from CentOS Stream. That gives it room to deviate from RHEL. It keeps drivers for hardware RHEL dropped, and AlmaLinux 10 ships an extra x86-64-v2 build for older CPUs (Nehalem era, roughly 2008 onward).
- **Rocky Linux** (founded by CentOS co-founder Gregory Kurtzer, governed by the RESF) stays bug-for-bug with RHEL. Rocky 10 requires x86-64-v3 (Haswell, roughly 2013 onward), same as RHEL 10.

We are likely to see these two Linux distros continue to grow in the future. If a vendor certification or compliance requirement demands an exact RHEL build, go with Rocky. If you have older hardware, use AlmaLinux.

More tools = less reliability

Akhilesh and Pragya Keshap, *Designing the Decision Surface: Building Reliable Agentic Systems* (slides)

The core claim: an agentic system gets less reliable the more tools you give it, especially when those tools overlap or are ambiguous. Every connection is another decision the model has to make: which tool to call, which arguments are needed, what order to use tools in, which results matter, and when to stop. Intuition may tell us that more tools would give the AI system more options and richer context, improving functionality and usefulness. Results on the ground are contrary to this and show that decisions an AI system needs to make compound. If the model gets each one right 95% of the time, a run that needs nine of them succeeds **only 63% of the time** (0.95^9)!

The fix starts before any code. We must understand the problem we're trying to solve deeply, define it well, and separate out the steps that are deterministic.

Domain tool abstraction

The main agent should not talk to dozens of low-level tools. Their example was a late delivery. Instead of exposing nine tools (customer lookup, orders, shipment, carrier events, incidents, refund and replacement eligibility, policy search), expose one `investigate_delivery_issue()` capability. Software owns the standard sequence, retries, and policy checks and returns a single structured evidence object. The model picks the capability, explains the result to the user, and asks for approval before doing anything of consequence.

Where reasoning belongs

Put algorithmic or rule based work in software and use a model only where input has to be interpreted:

- **Known record:** API or database lookup
- **Exact calculation or fixed policy:** code or a rules engine
- **Messy unstructured text:** a bounded model call with a typed output
- **Open-ended explanation:** the main model
- **High-impact exception:** a human

Evaluate each layer and not simply the final answer. A polished response can still incorrectly promise a refund the eligibility check rejected. Checks at routing, evidence retrieval, rules, and grounding tell you where something broke, instead of having to resort to tweaking the prompt. For trying prompts and comparing models before wiring them in, Google AI Studio has a Playground (link) and a Compare mode that runs the same prompt across models side by side.

The ad economy runs on fraud

Don Marti, *Keynote: Surviving the Rugpull Economy*

> Real markets have information symmetry.
> Don Marti

The keynote was about how ad tech quietly funds scams. The targeting algorithms optimize for one thing: putting the most lucrative ad on the cheapest page, and the cheapest pages are often scam sites. Ad tech also hands user IDs to bots, so advertisers pay for impressions no person ever saw. Fake and bad ads drive up costs for legitimate advertisers, and consumers end up on the receiving end of the scams. In his framing, fraud isn't an edge case. It is a load-bearing component of the business.

Google and Meta say that at their scale they can't police it. That excuse doesn't hold up to scrutiny since they move fast when it matters to them (immediate response to complaints from the International Olympic Committee was his example). It's like Burger King saying "We have so many Burger Kings we can't possibly mop all of the restrooms."

```
Economic signal + content > time + $ + risk
```

For ad supported pages to be worth it, the signal the ad carries plus the value of the content has to outweigh what the reader puts in. When ad money goes to the least expensive site, the medium can't build a reputation, the ad stops meaning anything, and product quality drops with it.

He reframed privacy by stating that it really isn't personal - it's a collective problem. Your own settings don't do much when the system runs on data from everyone else. Instead of relying on individual employees to install an adblocker, companies should automatically push one to all browsers. All it takes is one employee being tricked to click on a scam Homebrew link, and Binance is out of 6.3B dollars! The record $6.3B Binance heist started with a fake Homebrew install on Mac.

Local AI as a `systemd` service

David Duncan, *RamaLama + Podman Quadlets: Build Task-Specific AI Services with systemd*

RamaLama runs AI models in containers the way Podman runs everything else. It detects your GPU (falls back to CPU), pulls an inference image built for that hardware, pulls the model from Hugging Face, Ollama, or an OCI registry, and serves it. Models get treated like container images, and you never touch the host's GPU stack.

The workshop used one model server, with lightweight agent containers in front of it. Each agent gets its own system prompt and restart policy, isolated from the others. Quadlets turn each piece into a regular `systemd` service that starts on boot, restarts on failure, and logs to `journald` with no Kubernetes required.

For setting up a cluster, RamaLama can generate both a Quadlet and Kubernetes YAML, and Podman can run Kubernetes YAML through a `.kube` Quadlet. The same service moves between a single host and a cluster in either direction.

Dev environments are infrastructure. Is Cloud development the future?

Coder, *Beyond the Laptop: Building Software with Coder*

Coder moves development off local laptops and into cloud workspaces running on your own infrastructure, which also simplifies collaboration. Dev environments are treated like any other managed infrastructure and workspaces are defined as Terraform templates and provisioned onto Kubernetes, VMs, or containers, so the environment lives in version control and gets reviewed like code instead of drifting on every individual laptop.

I see the value for teams, especially onboarding and keeping environments consistent. Even an hour or two per developer spent setting up an environment adds up to tens of thousands of lost hours across a large engineering org. Instead of spending a week installing C++ dependencies and matching compiler versions on a new laptop, a developer clicks "Launch" and has a working, pre-built workspace in about a minute. There are limitations for hardware-dependent or low-level work like C, and abstracting the setup away means we lose some understanding of the stack. Setting up my own environment is how I learn, so this goes further than I'm comfortable with personally. *That said, I suspect this is the future for 80%+ of software engineering.*

RPM is still the foundation

Carl George, *RPM Packaging Essentials*

Flatpak, Snap, and AppImage get most of the attention, but native packages are still how software gets into the base OS and into repos like EPEL. Carl, who leads the EPEL team at Red Hat, ran a hands-on lab: write a spec file, build the package, inspect and install it, then run quality checks against it.

Here are two nifty commands to deal with spec files: `rpm --eval '%{_bindir}'` expands a single macro, and `rpm --showrc` shows every macro defined on the system.

OpenShift on homelab hardware

Thomas Cameron, *Enterprise Kubernetes at Home: A Homelabber's Guide to OKD*

OKD is the upstream community distribution of OpenShift, and running it at home or in a lab is a cheap way to build skills that transfer directly to the commercial product. The limitation is that most don't have hardware load balancers, storage arrays, or unlimited RAM. The infrastructure prerequisites are where most installs fail:

- **DNS:** a wildcard subdomain for application routes and correct reverse DNS pointer (PTR) records for node discovery
- **Load balancing:** HAProxy in front of the API and ingress
- **Storage:** local LVM for a single node, NFS exports for multi-node
- **Topology:** full HA, a hyper-converged virtualized cluster, or Single Node OKD (SNO), depending on your hardware and power budget

---

Models, package formats, and orchestration layers keep changing, but understanding the system and owning what gets merged is still our job.