Rethinking Auto-Scaling for Elixir Applications with Flame

Autoscaling is the idea that software systems can dynamically adjust their compute resources based on variable demand. It has long been a staple of modern web development. However, as evidenced in Chris McCord's talk, conventional notions of autoscaling carry significant baggage inherited from the days of platforms like Heroku. In a concurrency, fault-tolerance, and distribution-first framework like the Elixir and Phoenix ecosystem, these traditional approaches to scaling by spinning up more "web server" instances are generally either largely inadequate or overly complicated. McCord suggests a radical rethink of autoscaling using a new framework called Flame for elastic scale: eliminating problems rather than just solving them.

In the talk, McCord walks through the historical context shaping the current autoscaling paradigms, presents the limitations and complexities of present day serverless solutions and demonstrates with practical examples how Flame brings to Elixir a new form of more granular elastic scaling. Flame lets developers write code as if it were running on a local machine, provisioning and executing these workloads on demand across a cluster of ephemeral servers (essentially "fleeting Lambdas" that run the entire Elixir application anywhere, at any time, with minimal overhead) by leveraging the inherent distribution and concurrency capabilities of the Beam (Erlang Virtual Machine).

McCord begins by contextualizing the notion of autoscaling. He recalls the early 2000s, when deploying a web application meant dealing directly with physical servers and sales calls to hosting providers. Gradually, services like Heroku emerged, abstracting away the complexity of

provisioning servers and allowing developers to push code and have it run seamlessly in the cloud. Autoscaling quickly became synonymous with simply adding or removing Heroku "dynos" in response to traffic or workload demands. While convenient, this approach introduced an oversimplification: just run more web servers.

This "scale at the top level" mindset often leads to inefficiencies. For one, scaling the entire web tier just because a few CPU-intensive tasks bog down the system is costly and places unnecessary load on all parts of the infrastructure. Moreover, the load balancer indiscriminately routes all requests, both trivial and complex, across these instances. The result is a lack of fine-grained control, where sensitive operations like video transcoding or CPU-heavy encoding tasks are placed on the same servers handling critical API responses.

To illustrate a more ideal approach, McCord points out that Elixir applications can scale vertically and horizontally much better than many traditional languages. The Beam excels at concurrency and fault tolerance, making it possible to separate expensive tasks from the web tier. For example, one might run the web layer on a lightweight node and place CPU-intensive operations such as video encoding on a separate node. Still, horizontally scaling that video-processing node remains challenging. Current solutions might involve Kubernetes, serverless functions, or proprietary services, all of which add complexity and require the developer to learn new paradigms or integrate with multiple external systems.

What developers truly want, McCord argues, is not just "autoscaling" but elastic scaling at a granular level. Rather than scaling the entire application uniformly, it should be possible to scale

just the part of the system that is resource intensive. If a particular function or operation spikes in demand, you should be able to elastically provision more resources just for that function and then scale them back down as needed. Ideally, this would involve writing code once, on your laptop, and having it automatically scale elastically without rewriting the architecture or rethinking the entire deployment model.

The serverless model of AWS Lambda and similar services does attempt to solve this problem. By running code without the explicit management of servers, Lambda and its ecosystem allow developers to pay only for what they use, providing highly granular scaling. But as McCord highlights, this approach forces the use of proprietary ecosystems and complex chains of services. He gives a vivid example: a reference architecture for encoding video on AWS involves a labyrinth of services—S3 for storage, multiple step functions for orchestration, a proprietary Elemental Media Convert service for the actual encoding, CloudWatch for monitoring, SNS and SQS for message passing, and so forth. While Lambda solves the elastic scaling problem, it does so by introducing complexity, cost, and a difficult testing and development story. Instead of removing complexity, it merely shifts it around and locks the developer into a specific cloud provider's ecosystem.

In response, McCord presents Flame, a library that provides elastic scale within the Elixir ecosystem without rethinking the entire architecture or introducing a swarm of proprietary services. The acronym stands for Fleeting Lambda Application for Modular Execution, encapsulating the idea of ephemeral compute that can be spun up anywhere, on-demand, yet still run the developer's entire Elixir application.

Flame builds on the Beam's distributed capabilities. Rather than isolating a single function in a cold, stateless environment, Flame launches entire application nodes that contain your full supervision tree, database connections, Pub/Sub, and more. You can then call arbitrary blocks of code including closures that might contain database inserts, filesystem operations, or even shelling out to ffmpeg on these ephemeral nodes. When finished, these nodes simply disappear, and you stop paying for them. It's elastic computing that is invisible and matches the mental model of writing regular Elixir code locally.

The magic behind Flame comes from the Beam's distributed messaging and function calls. You wrap a block of code in *Flame.call* and Flame checks if there is currently a pool of nodes running that could execute that code. If not, it requests a new machine from your infrastructure provider (Fly.io for instance) spins it up with your application's Docker image and connects it back to the parent node. Distributed Erlang takes care of everything else. It ships the function and its captured state to the remote node, executes it, and returns the result as if it all happened locally.

One of the remarkable aspects McCord highlights is how little code Flame itself requires. Leveraging the Beam's native capabilities like *:erpc* and :*rpc*, and *Node.spawn_monitor*, the complexity melts away. Flame does not need custom serialization logic or rewriting of code. State, closures, and messages are transmitted seamlessly.

McCord provides several demos to illustrate Flame's power and simplicity. In a LiveView scenario, a user uploads a video file, and ffmpeg processes it frame-by-frame to produce

thumbnails. Traditionally, this might require a dedicated background node or a complicated serverless pipeline. With Flame, McCord simply wraps the process in *Flame.place_child*. The code runs on a temporary node, streams back thumbnails as they are generated, and the main application receives these messages as if they came from a local process. Testing remains straightforward since, in development mode, Flame just executes the code locally.

Another demo involves using the Bumblebee machine learning framework and Whisper models for transcription. Typically, ML workloads are resource-heavy and may require GPU resources or large models. By placing these tasks in Flame calls, the system can scale out as needed, running on appropriate hardware in different regions. While cold starts (i.e., the time to provision and load large ML models) can take tens of seconds, Flame still reduces complexity. You can even run these transcriptions concurrently while sending updates back to a LiveView, all without re-architecting the application around external queues or functions.

McCord also shows how Flame can run headless Chrome to measure webpage load times around the world. Instead of running Chrome locally, Flame calls let you spin up Chrome instances across multiple regions to test page speed from many vantage points. Results stream back to the main application no matter how many ephemeral nodes are spawned to handle the workload. Again, only one line of code changes to leverage Flame's power.

Flame runs your complete Elixir application without locking you into proprietary ecosystems. You write normal Elixir code, use your preferred database, broadcast via Phoenix Pub/Sub, and run system commands. In tests, it defaults to local mode, so everything works like a normal

inline call, simplifying development and deployment. Its pluggable backends already include Fly.io and Kubernetes and more can be added.

Though spinning up a node takes a few seconds, especially for CPU-bound or ML workloads, you can mitigate this by keeping a node warm or adjusting timeouts. Flame's configuration options let you fine-tune pools, concurrency, and idle shutdown times.

McCord's talk advocates a Beam-first approach to scaling. Instead of scaling entire apps or relying on vendor-driven serverless solutions, Flame allows granular, on-demand scaling at the function or process level. It aligns with Elixir's philosophy of simplicity and fault tolerance, letting developers focus on features, not infrastructure.