Terminal Velocity

Lessons learned from server scaling during the launch of Fall Guys

Who Am I?

Edwin Jones

Senior Software Engineer @

Mediatonic/Epic Games since 2017

Previously at EA

What’s this talk about?

- **What are Backend Game Services?**
- **How do we use them?**
- **How did this work on Fall Guys during our launch and beyond?**

What are Backend Game Services?

- Profile Storage
- Player Inventory
- Purchase Management
- Matchmaking
- Login Authentication
- And so on…

How are they architectured?

- **We use kubernetes/k8s to host our actor based containerised services.**
- **We use NATS.io as a communication layer between these components.**
- **We use cloud based queue tech to store important messages in a non volatile manner.**

How do we write services?

- Written in modern C#/.NET
- Primarily for compatibility with our unity client so we can share data transfer object definitions and simulation code.
- Targeting Dotnet Standard 2.0 in shared libs allows us to move ahead of where unity was/is in terms of feature support.

How do we host services?

- Hosted externally via Docker containers and Kubernetes (an open source container orchestration framework).
- Dotnet being platform agnostic allows us to deploy all services as normal linux containers.
- Kubernetes allows us to abstract (some) cloud provider details out of our CI/CD flows.

How do we host services?

- Containers are packaged applications and their dependencies which are sandboxed from other applications.

- Kubernetes/K8s uses an abstraction called pods to encapsulate and operate multiple containers.

How do we host services?

- Pods are the smallest “object” the Kubernetes api generally allows users to interact with.

- Pods are hosted on “nodes”

- Nodes are basically VM instances that host many pods, pods host one to many containers, each container generally hosts one instance of an application.

Credit: opsramp.com

Kubernetes Architecture

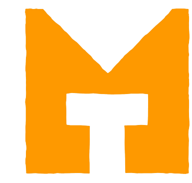
UI
User Interface

CLI
Kubectl

Kubernetes Master
API Server
Scheduler
Controller - Manager
etcd

Worker Node - 1
Pod-1
Container 1
Container 2
Container 3
Pod-2
Container 1
Pod-3
Container 1
Container 2
Kubelet
Kube-proxy

Worker Node - 2
Pod-1
Container 1
Container 2
Pod-2
Container 1
Container 2
Container 3
Pod-3
Container 1
Kubelet
Kube-proxy



How do we communicate with services?

- Via bidirectional message protocols.
- Protobuf as a transmission format for efficiency and ease of use in dotnet.
- Connections are “stateful” in that a handshake and JSON Web Token must be presented and refreshed.

How do we test services?

- Three layers of tests – Unit, Integration, Acceptance.
- Unit: Test a single class/method.
- Integration: Test combinations of classes in memory.
- Acceptance: Test all systems on a development environment via mocked clients.

How do we scale services?

- Horizontally via k8s deployments.
- Service infrastructure loosely coupled to node count and resources, change in node type rarely required outside of prep work for major launch load.
- Not all systems scaled as easily as others, we'll cover this later.

How did this work on Fall Guys?

Pretty Darn Well!
(All things considered...)

Launch Expectations

- 80k Peak CCU.
- We could handle 2x the predicted load.

What Broke?

- **650k CCU in first week!**
- **This was ~8x our predicted load, so 4 times higher than our 2x safety margin.**

How did it Break?

First, let’s go over how our services work

What do we use for Cloud Hosting?

Azure!

Why did we choose Azure?

- We were managing our own K8s deployments and wanted to move away from running our own clusters.
- Azure were one of the first providers to offer a mature offering that met our needs.
- Azure has great database/messaging offerings like Cosmos DB and Event Hubs.

Content Management System

CDN

HTTPS

HTTPS

Matchmaking System

WSS

Game Client

Game Servers

HTTPS

WSS

WSS

WSS

MM Gateway

Login Gateway

Player Gateway

Game Server GW

Game Server Allocation API

Fall Guys Services

Player Store

All Services → elasticsearch

All Services → DATADOG

- We use datadog for metrics and overall system performance monitoring
- We used kibana for easy searching of individual request logs per application for fine grained analysis or errors and bugs

What are Actors?

- Actors are a form of encapsulation of in memory state.
- Actors can only alter their state by receiving messages and they can also send messages to themselves and other actors.
- Actors don’t support concurrency internally, rather they process messages in a synchronous order allowing safer and easier business logic authoring.
- Actors facilitate scale - need more threads? Create more actors.
- Good for abstracting things like player profiles, game lobbies etc where you don’t need high concurrency at an instance level.

Why did we choose the Actor pattern?

- Primarily to chosen to provide thread safety in high concurrency situations.
- Reduces mental load during feature implementation.
- Worked really well with event sourcing for storage due to in memory actor state providing “free” caching.

How did we host our actors?

- We hosted many actors on one container.
- We used a custom partitioning system to decide which pods should host which player actors.
- We used consistent pod names as identifiers as part of this system.

What problems did we have?

- We wrote our own actor framework as leading C# options at the time didn’t support k8s or non windows deployment (Akka, Orleans etc)
- This took time and added work/complexity.
- We had to write our own partitioning system which caused knock on problems we will discuss later. We needed this to distribute load across hosts.
- Due to in memory state, memory usage was high in an individual pod as state had to be loaded at initialization rather than retrieved just in time from external stores.

Would we use actors again?

- Yes, the actor pattern trade offs were worth it in the long term.
- We do prefer stateless services for small implementations of new features and would not recommend actors for everything.

Content Management System

CDN

HTTPS

HTTPS

Matchmaking System

WSS

Game Client

Game Servers

HTTPS

MM Gateway

WSS

Login Gateway

WSS

Player Gateway

WSS

Game Server GW

Game Server Allocation API

Match Actor

Lobby Actor

Content Actor

Player Actor

Fall Guys Services

All Services → elasticsearch

All Services → DATADOG

Player Store

How was it all “interconnected”?

NATS.io!

What is NATS?

- **An open source messaging system.**
- **Designed for distributed cloud infrastructure.**

Why did we choose NATS?

- Gave us support for bi-directional messaging.
- Ran easily on Kubernetes.
- Easy to send protobuf messages between services and maintain a similar serialization stack as used in other parts of our systems.

What does that all look like put together?

Content Management System

CDN

HTTPS

HTTPS

Matchmaking System

WSS

Game Client

Game Servers

HTTPS

MM Gateway

WSS

Login Gateway

WSS

Player Gateway

WSS

Game Server GW

Game Server Allocation API

Match Actor

NATS

Lobby Actor

Content Actor

Player Actor

Fall Guys Services

Player Store

All Services → elasticsearch

All Services → DATADOG

What Broke *Really*?

We hadn't implemented proper circuit breaking for external systems

Matchmaking Overload

- Our matchmaking system was externally hosted and couldn’t immediately scale to our requirements.
- We made outages worse as we contributed to the load causing external systems to require more scaling and more time to cope.
- The player impact of this was that users couldn’t matchmake or play the game.
- We added exponential backoff logic and circuit breakers in within a week as this was a top priority for us.
- We used the C# library “Polly” to do most of the wiring and went with a “close the circuit for a few seconds if error rate breaches a pre configured value” approach.

What is exponential backoff?

- It’s a pattern where when you encounter an error in an external system you wait a set period of time before trying again to give it time to recover if it’s under load or experiencing a transient fault.
- The time between each failure grows exponentially (say to the power of 2) so the first delay is 2 seconds, then 4, then 8 etc. until you receive a successful response.
- There’s generally some upper limit to avoid the backoff growing to absurdly large numbers.

What is a “circuit breaker”?

- It’s a pattern where when you encounter an error in an external system and it keeps failing after several attempts you stop trying to call the external system.
- This stops your code wasting time running operations that are almost certain to fail.
- This also prevents you from continuing to strain the external system that has proven it cannot recover from whatever situation it has encountered.

What Broke *Really*?

We gave ourselves our own "Thundering Herd"

Thundering Herd

- We had configured clients to check the store daily at the same time in UTC.
- This meant our own clients were DDoSing us and denying people access to the store.
- We implemented jitter to fix this quickly.
- We didn't need to implement exponential backoff as this resolved the issue.

What Broke *Really*?

NATs did not initially scale with our designs

NATs scaling

- We'd configured nats to create a response topic for every request topic for bi-directionality. Due to our scale this lead to millions of topics being created. This overwhelmed our initial nats hosts.
- We tried to scale out of this horizontally but we couldn't copy millions of topic entries effectively to the new nodes.
- We eventually fixed this by updating our NATs version and switching to subscription rules to reduce the amount of topics.

What Broke *Really*?

We had head of line blocking for reward processing

Head of line blocking

- Our reward flow uses partitioned, concurrent queues. This didn’t matter when NATS issues backed up our systems as there was nothing able to dequeue the messages.
- This meant rewards didn’t get processed at the end of games for users and lead to high amounts of support ticket requests and unhappy players.
- Resolving the NATS issues allowed reward flows to slowly release pressure and over time users got their rewards. Our queues allowed strong recovery as data wasn’t lost, just stuck until we fixed the other systems.

Game Servers

Azure Sevice Bus

Game Server Gateway

Episode Processor

Player Episode Processor

NATS

Player Actor

Fall Guys Services

Conclusions from Launch

- We were able to quickly scale up with Kubernetes for problems that could be solved with horizontal scaling.
- Most other problems could be solved with traditional fixes like circuit breakers, exponential backoff and jitter implementations.
- Nats was tricky but ultimately issues were resolved by a simple version upgrade - the architectural choice was proven to be sound.
- Using queues meant that even though things broke, no important data was lost.

Post Launch (2021-2022)

- Cross Progression
- Season Pass
- Squads Mode
- And more!

But then, suddenly in 2022....

FALL GUYS
SEASON 1
FREE FOR ALL

How we'd prepped

- Extensive Load Testing (for ~3M CCU)
- Scaled up in Azure
- Informed partners of expected load
- We'd written a three part load test system - local CLI, remote controller and scalable remote tester pool.
- Remote test services were hosted in their own K8s cluster we could scale independently of our dev and prod environments.
- We didn't catch anything new this time but we would advise scaling higher than your initial assumptions to flush out weaknesses you may not see at lower loads - this would have saved us a lot of pain in 2020!

What Broke?

- We hit 20 million users in the first 48 hours
- Our game server provider’s allocation API couldn't scale fast enough
- Our login systems couldn’t handle the load

How does our login work?

- All game clients have to present some form of credentials for their platform - PSN/Steam/Epic Store etc.
- We use third party apis to confirm these credentials.
- We generate a signed token with a time based expiry for the client to use once we've verified the user is authentic. Clients will periodically refresh these before they expire outside of normal game play.
- Clients can then use this token to send authenticated messages to other systems such as the player gateway, matchmaking system and so on.

How did we fix it?

- We had to throttle login requests.
- We optimised our game server allocation system.
- We scaled up all the things.
- We didn’t have an existing throttling system so we built one from scratch.
- We fixed game server allocation by batching requests - load testing didn’t catch this.
- We had to scale CosmosDB more than the azure portal would allow us by default and had to submit a special out of band request to Microsoft.

Conclusions from F2P Launch

- Load testing didn't catch the issues we faced mostly because of the size of the real scale vs expectations and limits with external systems we hadn't been aware of previously.
- Despite this, most systems scaled horizontally as expected and we'd learned most of our lessons from the initial launch.

What did we learn?

- **Stateful sets in k8s can lead to long deployment times with hundreds of pods.**
- **Long deployments lead to back pressure.**
- **Back pressure leads to delays.**
- **Delays lead to *suffering!***

What did we learn?

- We needed stateful sets for consistent pod naming due to how our actor partitioning systems worked.
- At the time, these worked with a rolling update in series rather than in parallel.
- We had ~400 player profile pods to update at a time.
- This took a long time! We wanted updates to take a few minutes but they could take ~½ hour or more.

What did we learn?

- We didn't change this immediately and accepted that updates to this service would take longer.
- This did mean production hotfixes took longer to deploy than we would have liked.
- There was no obvious impact to players as pods were only down for seconds for each update and had consistent naming when they came back, meaning they would take the same partition.
- If we had encountered issues we would have looked at alternate update strategies and overhauled our partitioning system to handle them (such as pod availability budgets for concurrent updates etc.)

Wrapping up…

- **Actor systems provide a strong, comprehensible and scalable model for concurrency.**
- **Scaling is hard but Kubernetes gives you control by giving you fine grained control over how many instances of a service you have, how you deploy them, how you manage versioning them and rolling them back with tools like Helm.**
- **Load testing is crucial before a launch but you can't test it all. Test more than you think you will need to and avoid missing flaws when you've got the time to fix them.**
- **You can prepare for this pain and make your life easier by reading up about common problems (thundering herd, circuit breakers) and implementing solutions them into your systems *before* you need them.**

Wrapping up...

- Quick and free guide to Kubernetes:
  https://kube101.jeffgeerling.com/
- What is event sourcing:
  https://arkwright.github.io/event-sourcing.html
- Actor pattern made simple:
  https://blog.grio.com/2021/03/introduction-to-the-actor-model-using-real-actors.html
- NATS.io overview:
  https://docs.nats.io/nats-concepts/overview

Thank you!

Bio: https://edwinjones.me.uk

Twitter: @edwinjonesuk