WEBVTT

1
00:00:00.900 --> 00:00:03.700
<v ->What happens when you have greater cloud choice?</v>

2
00:00:03.700 --> 00:00:07.310
With VMware cloud on AWS, all your apps can benefit

3
00:00:07.310 --> 00:00:11.340
from a cloud that gives you choice, flexibility, and speed.

4
00:00:11.340 --> 00:00:13.703
VMware, welcome change.

5
00:00:15.258 --> 00:00:18.675 line:15%
(relaxing music playing)

6
00:00:25.540 --> 00:00:29.840
<v ->Hi, welcome to beyond five 9s,</v>

7
00:00:29.840 --> 00:00:31.790
which is all about lessons from our highest

8
00:00:31.790 --> 00:00:33.170
available data planes.

9
00:00:33.170 --> 00:00:35.920
I'm Colm, I'm a VP and distinguished engineer here

10
00:00:35.920 --> 00:00:39.704
at Amazon web services, we're gonna be joined by Yasemin,

11
00:00:39.704 --> 00:00:41.950
who's a principal engineer at Amazon web services,

12
00:00:41.950 --> 00:00:43.800
she works on Kinesis.

13
00:00:43.800 --> 00:00:48.640
And today we're gonna share a bunch of insights into

14
00:00:48.640 --> 00:00:50.990
how we build our most highly available systems.

15
00:00:50.990 --> 00:00:53.330
You know, what's going on under the hood

16
00:00:53.330 --> 00:00:55.113
or behind the scenes, so to speak.

17
00:00:56.210 --> 00:00:58.170
You know, we've gone into great detail about

18
00:00:58.170 --> 00:01:01.690
some of these topics in our Amazon builders library series.

19
00:01:01.690 --> 00:01:05.310
But today we're gonna cherry pick some of the best

20
00:01:06.320 --> 00:01:08.880
lessons and condense them down into, you know,

21
00:01:08.880 --> 00:01:12.400
10 simple things hopefully you can take away and apply

22
00:01:12.400 --> 00:01:15.040
to your own systems as you build them.

23
00:01:15.040 --> 00:01:19.285
These, these lessons come from systems that have, you know,

24
00:01:19.285 --> 00:01:23.500
stood the test of time and had very good availability

25
00:01:23.500 --> 00:01:25.733

records, which is not easy to achieve.

26
00:01:27.290 --> 00:01:29.220
We've talked about this before,

27
00:01:29.220 --> 00:01:32.274
and some of the lessons are eternal and,

28
00:01:32.274 --> 00:01:33.860
and some of the lessons are new.

29
00:01:33.860 --> 00:01:35.730
But the first one that we're gonna start with is,

30
00:01:35.730 --> 00:01:39.370
is definitely one that's eternal, which is it's,

31
00:01:39.370 --> 00:01:41.840
it's all about insisting on high standards, right?

32
00:01:41.840 --> 00:01:45.184
Nothing we're gonna go into, there's no tip or trick

33
00:01:45.184 --> 00:01:48.740
or magic that we have for you that can ever replace

34
00:01:48.740 --> 00:01:52.600
paying very strong attention to detail in,

35
00:01:52.600 --> 00:01:54.743
in how we build systems.

36
00:01:56.260 --> 00:01:59.080
There's a few things I'm always reminded about

37
00:01:59.080 --> 00:02:01.040
when I think of this.

38
00:02:01.040 --> 00:02:02.870
One is something I learned from Werner Vogels,

39
00:02:02.870 --> 00:02:05.330
our CTO a long time ago, which is when,

40
00:02:05.330 --> 00:02:07.130
which is that when you have, you know,

41
00:02:07.130 --> 00:02:09.830
systems that are processing billions of requests,

42
00:02:09.830 --> 00:02:12.283
or you know, even trillions of requests over,

43
00:02:13.165 --> 00:02:15.850
you know, short periods of time,

44
00:02:15.850 --> 00:02:18.130
pretty much anything that could go wrong in the system

45
00:02:18.130 --> 00:02:21.620
will be going wrong at some point in time,

46
00:02:21.620 --> 00:02:25.110
some request somewhere is probably experiencing, you know,

47
00:02:25.110 --> 00:02:26.940
even a one in a billion failure, right?

48
00:02:26.940 --> 00:02:28.950
Because you have billions of requests.

49
00:02:28.950 --> 00:02:32.940
And so it requires paying enormous attention to detail,

50
00:02:32.940 --> 00:02:34.800

looking for every single error case,

51
00:02:34.800 --> 00:02:37.005
looking for everything that could possibly go wrong

52
00:02:37.005 --> 00:02:39.983
and having a plan for that in the code and in testing

53
00:02:39.983 --> 00:02:44.670
and everywhere, and that kind of culture, at team level,

54
00:02:44.670 --> 00:02:46.710
there's just no substitute for it.

55
00:02:46.710 --> 00:02:50.710
You know, having teams who are willing to constantly raise

56
00:02:50.710 --> 00:02:53.760
the bar on testing, constantly, you know, write new

57
00:02:53.760 --> 00:02:56.780
and interesting tests or come up with ways to make

58
00:02:56.780 --> 00:02:58.330
testing itself easier.

59
00:02:58.330 --> 00:03:01.900
You know, a lot of that is where the quality

60
00:03:01.900 --> 00:03:03.837
and high availability in our systems comes from,

61
00:03:03.837 --> 00:03:07.310
and like I say, there's no substitute for it.

62
00:03:07.310 --> 00:03:11.730
And I often think that that kind of quality comes about

63
00:03:11.730 --> 00:03:13.060
from great habits, right?

64
00:03:13.060 --> 00:03:16.090
It's from what we do over and over and over again.

65
00:03:16.090 --> 00:03:19.177
And it's really hard to approach a system and go, you know,

66
00:03:19.177 --> 00:03:20.870
"I'm just gonna add quality to it.

67
00:03:20.870 --> 00:03:23.890
I'm just gonna try to make it better than it was before."

68
00:03:23.890 --> 00:03:26.090
It's more like something pervasive that has to be there

69
00:03:26.090 --> 00:03:27.500
from the beginning.

70
00:03:27.500 --> 00:03:30.470
And so something we've been paying special attention to

71
00:03:30.470 --> 00:03:32.780
on this kind of a quality dimension for,

72
00:03:32.780 --> 00:03:37.780
for a long time now is deployment safety, where,

73
00:03:37.840 --> 00:03:41.560
you know, we write our code and we go through code review,

74
00:03:41.560 --> 00:03:42.660
collaborative code review,

75
00:03:42.660 --> 00:03:46.410

and so on and back and forth processes like you'd be very

76
00:03:46.410 --> 00:03:49.470
familiar with, but once we've checked in our code,

77
00:03:49.470 --> 00:03:52.530
we actually have quite sophisticated deployment systems

78
00:03:52.530 --> 00:03:55.900
that take over after that and kind of take charge

79
00:03:55.900 --> 00:03:57.930
of getting that code from, you know,

80
00:03:57.930 --> 00:03:59.370
the fact that it's been checked in,

81
00:03:59.370 --> 00:04:01.690
all the way to running in production.

82
00:04:01.690 --> 00:04:05.030
And that system has been designed in a way where it

83
00:04:05.030 --> 00:04:09.000
generates a lot of positive safety for our systems.

84
00:04:09.000 --> 00:04:11.650
You know, whenever we're deploying new code, right?

85
00:04:11.650 --> 00:04:13.530
That always means some risks.

86
00:04:13.530 --> 00:04:16.740
We try to be very paranoid about our testing.

87
00:04:16.740 --> 00:04:18.210
We try to be very creative in our testing

88
00:04:18.210 --> 00:04:21.230
and test it in every possible way, and I'll talk more

89
00:04:21.230 --> 00:04:23.570
about that later, but you know,

90
00:04:23.570 --> 00:04:26.030
new code that hasn't been run in production before,

91
00:04:26.030 --> 00:04:28.090
there's always some chance that it might encounter

92
00:04:28.090 --> 00:04:30.910
some condition that we didn't plan for.

93
00:04:30.910 --> 00:04:35.430
And so we've got, you know, CICD processes that make sure

94
00:04:35.430 --> 00:04:38.410
that, you know, without us having to be too involved,

95
00:04:38.410 --> 00:04:40.950
that code will go through, you know,

96
00:04:40.950 --> 00:04:44.870
pretty careful, deliberate testing and deployment

97
00:04:44.870 --> 00:04:46.050
as it goes, right?

98
00:04:46.050 --> 00:04:48.090
So we'll deploy it to, you know,

99
00:04:48.090 --> 00:04:52.040
maybe just one box to start with and see how it goes,

100
00:04:52.040 --> 00:04:55.200

there'll be some tests run and that box will have to pass

101
00:04:55.200 --> 00:04:57.790
all of those tests in every condition.

102
00:04:57.790 --> 00:04:58.870
And if it doesn't, you know,

103
00:04:58.870 --> 00:05:01.270
we've got fast and reliable rollback,

104
00:05:01.270 --> 00:05:03.640
that's obviously pretty important too,

105
00:05:03.640 --> 00:05:06.110
but if it does go well, you know, it gets promoted.

106
00:05:06.110 --> 00:05:07.580
And the idea is, you know,

107
00:05:07.580 --> 00:05:10.790
the more and more confidence we have in that code,

108
00:05:10.790 --> 00:05:15.710
the broader we're, we're willing to run that code, right?

109
00:05:15.710 --> 00:05:18.410
So, as we develop confidence in it because it's passing

110
00:05:18.410 --> 00:05:21.790
more and more tests, we'll promote it from just one box

111
00:05:21.790 --> 00:05:26.520
to one cell maybe, or one availability zone, one region,

112
00:05:26.520 --> 00:05:28.540
multiple regions and so on.

113
00:05:28.540 --> 00:05:31.080
And this is something, you know, we've gone really,

114
00:05:31.080 --> 00:05:32.320
really deep at our culture.

115
00:05:32.320 --> 00:05:35.190
You know, every development team at AWS knows

116
00:05:35.190 --> 00:05:39.440
and understands this deployment process pretty intricately

117
00:05:39.440 --> 00:05:42.010
and knows how to, you know, get the most out of it

118
00:05:42.010 --> 00:05:45.950
and design their deployments and code in a way where

119
00:05:45.950 --> 00:05:47.940
they're gonna extract the most benefit from it,

120
00:05:47.940 --> 00:05:49.800
and We've seen a lot of return from this.

121
00:05:49.800 --> 00:05:52.940
It's been, it's been, you know,

122
00:05:52.940 --> 00:05:56.430
a great mechanism for us to deliver

123
00:05:56.430 --> 00:05:58.500
high availability in systems.

124
00:05:58.500 --> 00:06:02.060
And, you know, the occasional issues that it catches

125
00:06:02.060 --> 00:06:03.490

are well, well worth it.

126
00:06:03.490 --> 00:06:04.890
So that's, that's our first,

127
00:06:05.780 --> 00:06:07.590
first thing, and the first example of insisting

128
00:06:07.590 --> 00:06:09.140
on high standards.

129
00:06:09.140 --> 00:06:11.810
The next, and something we haven't really talked too much

130
00:06:11.810 --> 00:06:16.270
about before is, is how we manage our systems, right?

131
00:06:16.270 --> 00:06:18.450
So there's, there's this concept out there, which,

132
00:06:18.450 --> 00:06:20.687
which really condensed it down for me, and I love about,

133
00:06:20.687 --> 00:06:23.850
you know, you want to manage your systems,

134
00:06:23.850 --> 00:06:26.780
more like cattle than like kept pets in that you want to be

135
00:06:26.780 --> 00:06:31.250
able to think of them in abstractions and not, you know, be,

136
00:06:31.250 --> 00:06:34.410
be managing, you know, literally one host or one server

137
00:06:34.410 --> 00:06:37.920
at a time, because that's not a very, very scalable,

138
00:06:37.920 --> 00:06:41.200
and actually most systems at AWS have kind of long been

139
00:06:41.200 --> 00:06:44.220
beyond the scale that can be managed by hand, you know.

140
00:06:44.220 --> 00:06:48.180
Even the systems at AWS when I joined 13 years ago,

141
00:06:48.180 --> 00:06:51.370
were already quite far beyond that scale.

142
00:06:51.370 --> 00:06:54.410
And we use, you know, today we use the same concepts

143
00:06:54.410 --> 00:06:56.170
you'd be familiar with to manage systems,

144
00:06:56.170 --> 00:06:59.530
we have auto scaling groups, VPC, subnets, security groups,

145
00:06:59.530 --> 00:07:00.870
all of these things.

146
00:07:00.870 --> 00:07:03.550
Internally, we have this concept called host classes,

147
00:07:03.550 --> 00:07:07.750
they're very similar, and they just let us manage, you know,

148
00:07:07.750 --> 00:07:10.710
collections of, of machines without having to think about

149
00:07:10.710 --> 00:07:13.540
the individual machines, and our deployment systems,

150
00:07:13.540 --> 00:07:16.240

you know, they work on top of that, they clone these entire

151
00:07:16.240 --> 00:07:18.403
abstractions between regions, for example.

152
00:07:19.450 --> 00:07:22.730
But we've actually been going much, much further, right?

153
00:07:22.730 --> 00:07:25.790
So, we've learned that we, you know, we have to be able to

154
00:07:25.790 --> 00:07:28.850
operate our systems in hands-off ways,

155
00:07:28.850 --> 00:07:33.850
both for safety, operational safety and for security.

156
00:07:34.120 --> 00:07:37.689
And typically, you know, AWS operators, you know,

157
00:07:37.689 --> 00:07:41.620
like me, as a developer, just simply don't have access

158
00:07:41.620 --> 00:07:44.170
to say, every AWS region.

159
00:07:44.170 --> 00:07:48.500
And so, I could never really plan on being able to log in

160
00:07:48.500 --> 00:07:51.140
or access the system to do something, it's just not our

161
00:07:51.140 --> 00:07:52.663
operational model at all.

162
00:07:54.188 --> 00:07:57.000
But then we have some systems, like AWS outposts

163
00:07:57.000 --> 00:07:59.710
and Snowball that are designed to be disconnected

164
00:07:59.710 --> 00:08:00.700
for periods of time.

165
00:08:00.700 --> 00:08:02.740
You know, we may just have no access to them,

166
00:08:02.740 --> 00:08:05.050
nobody may have any access to them for a while

167
00:08:05.050 --> 00:08:07.730
because of the nature of the product.

168
00:08:07.730 --> 00:08:10.830
And so, we've really had to double and triple down on the,

169
00:08:10.830 --> 00:08:12.900
the automation that it takes to be able to operate

170
00:08:12.900 --> 00:08:15.130
in that environment, to have systems that are self-healing

171
00:08:15.130 --> 00:08:17.593
and that can kind of take care of themselves.

172
00:08:19.160 --> 00:08:22.080
We have kind of two classes of systems these days.

173
00:08:22.080 --> 00:08:26.120
We've kind of got, we've got bastioned systems where we have

174
00:08:26.120 --> 00:08:29.980
very limited access to those systems via what we call

175
00:08:29.980 --> 00:08:32.180

bastion service or bastion hosts

176
00:08:33.500 --> 00:08:37.680
that let us, you know, recover systems typically if,

177
00:08:37.680 --> 00:08:40.750
if there were some urgent need to, or something like that.

178
00:08:40.750 --> 00:08:45.240
But with, you know, a strong record of the fact that folks

179
00:08:45.240 --> 00:08:48.871
have had to either do something with that system and,

180
00:08:48.871 --> 00:08:51.350
and notification processes for that and so on.

181
00:08:51.350 --> 00:08:54.940
But we also have systems where, increasingly that's just

182
00:08:54.940 --> 00:08:55.920
not an option.

183
00:08:55.920 --> 00:08:59.020
You know, there's no general purpose interactive

184
00:08:59.020 --> 00:09:02.900
administrative access, and even in the event of having to

185
00:09:02.900 --> 00:09:05.200
recover a system, there's,

186
00:09:05.200 --> 00:09:08.750
there's just simply no general purpose way to access it.

187
00:09:08.750 --> 00:09:11.680
And instead, we have to have built in in advance, you know,

188
00:09:11.680 --> 00:09:14.650
all the mechanisms that we would need.

189
00:09:14.650 --> 00:09:16.750
You know, the AWS Nitro system is a really strong

190
00:09:16.750 --> 00:09:18.910
example of that, and that's the, that's the system

191
00:09:18.910 --> 00:09:22.450
that runs, you know, our modern EC2 instance types.

192
00:09:22.450 --> 00:09:27.300
And, you know, there's simply no mechanism at all where,

193
00:09:27.300 --> 00:09:30.150
you know, I or someone else could go run a command

194
00:09:30.150 --> 00:09:31.900
on that system or,

195
00:09:31.900 --> 00:09:35.090
or access it in some kind of interactive way.

196
00:09:35.090 --> 00:09:37.640
Instead, everything's done via kind of pre-planned,

197
00:09:38.511 --> 00:09:40.860
pre secured, you know, fully authenticated

198
00:09:40.860 --> 00:09:42.810
and encrypted APIs, and that's the only way

199
00:09:42.810 --> 00:09:44.160
for things to happen.

200
00:09:44.160 --> 00:09:48.140

And, and we found, you know, great motivation for this

201
00:09:48.140 --> 00:09:50.430
is security, but this, this talk is about availability

202
00:09:50.430 --> 00:09:53.280
and a great benefit for availability too, is it just means

203
00:09:53.280 --> 00:09:56.450
there's no possibility of any untracked changes.

204
00:09:56.450 --> 00:09:59.200
And there's no possibility of, you know,

205
00:09:59.200 --> 00:10:01.460
somebody did something to fix something and then forgot

206
00:10:01.460 --> 00:10:02.850
about it or anything like that.

207
00:10:02.850 --> 00:10:04.940
It's just, it can't be that kind of system.

208
00:10:04.940 --> 00:10:07.160
Instead, it's much more hermetic.

209
00:10:07.160 --> 00:10:09.140
So that's our, that's our second item.

210
00:10:09.140 --> 00:10:11.500
I'm gonna hand over to Yasemin, who's gonna now tell us

211
00:10:11.500 --> 00:10:13.843
about the third item on our list.

212
00:10:15.210 --> 00:10:16.347
<v ->Thank you, Colm.</v>

213
00:10:17.650 --> 00:10:21.590
Well, while we design to avoid failures altogether.

214
00:10:21.590 --> 00:10:24.670
we also design with failures in mind.

215
00:10:24.670 --> 00:10:26.540
This is mainly because we,

216
00:10:26.540 --> 00:10:31.060
we know that failures may happen in rare cases,

217
00:10:31.060 --> 00:10:33.940
and you want to make sure that the blast radius

218
00:10:33.940 --> 00:10:38.690
of such events are as minimal as possible.

219
00:10:38.690 --> 00:10:42.670
I will discuss four techniques that we use for this purpose.

220
00:10:42.670 --> 00:10:47.120
The first technique we use is regional isolation.

221
00:10:47.120 --> 00:10:52.100
AWS cloud spans 25 geographic regions around the world,

222
00:10:52.100 --> 00:10:55.020
and there will be eight more regions that are announced

223
00:10:55.020 --> 00:10:56.203
to be launching soon.

224
00:10:59.540 --> 00:11:03.120
Each one of these region is isolated from each other,

225
00:11:03.120 --> 00:11:06.773

both geographically, as well as on the software stack.

226
00:11:07.760 --> 00:11:11.030
AWS services offer regional end points,

227
00:11:11.030 --> 00:11:14.350
enabling direct access to that region.

228
00:11:14.350 --> 00:11:19.330
Using regional isolation, any rare failure that may come up

229
00:11:19.330 --> 00:11:22.940
in one region stays in that region.

230
00:11:22.940 --> 00:11:27.610
Say for example, we have an event going on in us-west-1,

231
00:11:27.610 --> 00:11:29.825
it will be contained within the region,

232
00:11:29.825 --> 00:11:33.523
and will not spread to any of the other regions.

233
00:11:36.310 --> 00:11:40.060
Next technique we use is zonal isolation.

234
00:11:40.060 --> 00:11:43.960
Each AWS region offers multiple availability zones

235
00:11:43.960 --> 00:11:46.217
within it's periphery.

236
00:11:46.217 --> 00:11:50.360
Availability zones are data centers that are miles away

237
00:11:50.360 --> 00:11:51.193
from each other.

238
00:11:52.240 --> 00:11:56.360
AWS cloud has 81 availability zones across the world,

239
00:11:56.360 --> 00:11:59.343
with 24 more launching soon.

240
00:12:00.800 --> 00:12:04.963
Similar to regions, AZ's are also isolated from each other.

241
00:12:05.877 --> 00:12:09.150
A failure in one AZ, let's say us-west-2a

242
00:12:10.020 --> 00:12:12.790
is having an event, will not spread

243
00:12:12.790 --> 00:12:14.043
to any of the other AZ's.

244
00:12:16.740 --> 00:12:19.550
One example where we leverage this property

245
00:12:19.550 --> 00:12:21.550
is regional services.

246
00:12:21.550 --> 00:12:26.200
There will be multiple availability zones serving traffic,

247
00:12:26.200 --> 00:12:28.670
routed to regional end points.

248
00:12:28.670 --> 00:12:32.760
This way, in the event of availability zone failures,

249
00:12:32.760 --> 00:12:36.570
traffic will be redirected to healthy zones

250
00:12:36.570 --> 00:12:40.690

behind the region, and customers will not be serving,

251
00:12:40.690 --> 00:12:43.560
will not be experiencing any degradation.

252
00:12:43.560 --> 00:12:47.290
So far, we discussed physical compartmentalization

253
00:12:47.290 --> 00:12:52.290
of architecture by using AWS regions and availability zones.

254
00:12:52.410 --> 00:12:55.480
These two have regional and zonal blast radius

255
00:12:55.480 --> 00:12:56.713
impact accordingly.

256
00:12:58.640 --> 00:13:01.510
We apply cellular isolation to further reduce

257
00:13:01.510 --> 00:13:04.020
the blast radius impact of events.

258
00:13:04.020 --> 00:13:07.540
In this technique, we build dedicated software stacks

259
00:13:07.540 --> 00:13:09.320
which we call cells.

260
00:13:09.320 --> 00:13:11.860
Cells are isolated from each other,

261
00:13:11.860 --> 00:13:14.103
with their own dedicated end points.

262
00:13:15.370 --> 00:13:18.270
Customers get assigned to one of these cells.

263
00:13:18.270 --> 00:13:21.300
Imagine there are eight customers and four cells,

264
00:13:21.300 --> 00:13:23.693
we have two customers assigned to each of them.

265
00:13:24.840 --> 00:13:27.370
Since cells are isolated from each other,

266
00:13:27.370 --> 00:13:31.210
failure in one cell, let's say cell two is having an event,

267
00:13:31.210 --> 00:13:33.870
will not spread to any of the other cells.

268
00:13:33.870 --> 00:13:36.750
That means it's only the customers assigned to that cell,

269
00:13:36.750 --> 00:13:40.350
the palm tree and stars in this case, will be impacted,

270
00:13:40.350 --> 00:13:43.063
but no other customers will be impacted.

271
00:13:44.240 --> 00:13:45.890
This is pretty good, right?

272
00:13:45.890 --> 00:13:50.000
It's much better than a zonal and regional blast radius

273
00:13:50.000 --> 00:13:51.940
that we discussed before.

274
00:13:51.940 --> 00:13:53.093
But can we do better?

275
00:13:54.280 --> 00:13:58.050

We do better by using shuffle sharding.

276
00:13:58.050 --> 00:14:01.730
In shuffle sharding, we divide the service into smaller

277
00:14:01.730 --> 00:14:05.230
compartments, which I will call partitions in this case,

278
00:14:05.230 --> 00:14:06.473
the blue boxes.

279
00:14:07.624 --> 00:14:11.680
And we assign customers to partitions, not one to one,

280
00:14:11.680 --> 00:14:14.630
but we assign multiple, multiple partitions

281
00:14:14.630 --> 00:14:16.270
to a single customer.

282
00:14:16.270 --> 00:14:19.700
In this case, I have two partitions assigned

283
00:14:19.700 --> 00:14:21.760
to each customer.

284
00:14:21.760 --> 00:14:25.130
So, let's evaluate the blast radius impact of failures

285
00:14:25.130 --> 00:14:26.650
in this case.

286
00:14:26.650 --> 00:14:30.690
Let's say partition one is having an event.

287
00:14:30.690 --> 00:14:34.620
In that case, both palm tree and stars have another

288
00:14:34.620 --> 00:14:36.540
partition that's healthy.

289
00:14:36.540 --> 00:14:41.030
Therefore they will not see impact and they will continue to

290
00:14:41.030 --> 00:14:42.293
operate just fine.

291
00:14:44.220 --> 00:14:47.980
It's only the time around both partitions of a single

292
00:14:47.980 --> 00:14:51.490
customer having an event then that customer

293
00:14:51.490 --> 00:14:53.350
will see the impact.

294
00:14:53.350 --> 00:14:56.980
And in this case, when partition four is also out,

295
00:14:56.980 --> 00:15:01.280
it's the stars that will see the impact because those two

296
00:15:01.280 --> 00:15:03.530
are shared partition for stars,

297
00:15:03.530 --> 00:15:07.793
but palm tree and hiker will not observe any impact.

298
00:15:09.600 --> 00:15:13.580
So there are two benefits that I like to highlight here.

299
00:15:13.580 --> 00:15:17.040
The real first one is creating customer impact

300
00:15:17.040 --> 00:15:19.670

is much harder with shuffle sharding.

301
00:15:19.670 --> 00:15:23.410
This is mainly because it takes multiple partitions

302
00:15:23.410 --> 00:15:26.253
to have failure, to create an impact,

303
00:15:27.110 --> 00:15:30.560
and probability of having a failure across multiple

304
00:15:30.560 --> 00:15:34.900
partitions is much lower than probability of having

305
00:15:34.900 --> 00:15:37.113
a failure on a single partition.

306
00:15:38.410 --> 00:15:42.920
The second benefit is that even in the amount of those very,

307
00:15:42.920 --> 00:15:46.110
very rare probability events that are happening

308
00:15:46.110 --> 00:15:50.396
across partitions, the impact created to the customer

309
00:15:50.396 --> 00:15:52.410
is much lower.

310
00:15:52.410 --> 00:15:55.630
We have one customer being impacted in the case of

311
00:15:55.630 --> 00:15:59.640
two partition failures compared to impacting multiple

312
00:15:59.640 --> 00:16:02.650
customers in the regular sharding schemes

313
00:16:02.650 --> 00:16:03.990
would actually create.

314
00:16:03.990 --> 00:16:06.820
Next, I will discuss circuit breakers.

315
00:16:06.820 --> 00:16:09.740
Circuit breakers is one of those techniques we used to

316
00:16:09.740 --> 00:16:13.373
eliminate failures, and I will discuss two examples.

317
00:16:14.410 --> 00:16:15.883
First one is load shedding.

318
00:16:16.720 --> 00:16:21.610
We know systems can slow down, or sometimes even fall over

319
00:16:21.610 --> 00:16:23.530
under excess load.

320
00:16:23.530 --> 00:16:26.320
We design to make sure that our services are not

321
00:16:26.320 --> 00:16:27.773
vulnerable to this problem.

322
00:16:28.860 --> 00:16:32.720
To address this issue, we first identify the maximum

323
00:16:32.720 --> 00:16:35.683
capacity of every individual component.

324
00:16:36.770 --> 00:16:39.980
We use stress testing to get this information.

325
00:16:39.980 --> 00:16:43.610

Now we install the load shedders locally within the services

326
00:16:43.610 --> 00:16:45.903
to monitor the traffic being served.

327
00:16:47.130 --> 00:16:49.130
Once service starts receiving traffic

328
00:16:49.130 --> 00:16:52.560
more than it's predefined maximum capacity limit,

329
00:16:52.560 --> 00:16:56.293
the load shedders start rejecting any excess load.

330
00:16:57.260 --> 00:16:59.920
They are designed to reject a load very quickly

331
00:16:59.920 --> 00:17:03.160
without spending much of the system's resources.

332
00:17:03.160 --> 00:17:06.900
This way we ensure that systems continue to operate

333
00:17:06.900 --> 00:17:09.193
successfully under excess load.

334
00:17:12.490 --> 00:17:14.930
The other circuit breaker that I'll discuss

335
00:17:14.930 --> 00:17:16.880
is bullet counters.

336
00:17:16.880 --> 00:17:20.460
This pattern gets used frequently on external monitoring

337
00:17:20.460 --> 00:17:24.160
applications, like Heart Guardians in this case.

338
00:17:24.160 --> 00:17:28.730
These applications are common across AWS because we want to

339
00:17:28.730 --> 00:17:32.910
make sure that we are the first ones to detect any problem

340
00:17:32.910 --> 00:17:36.520
that's maybe ongoing and mitigate it right away

341
00:17:36.520 --> 00:17:38.483
before customers are even noticing it.

342
00:17:40.150 --> 00:17:44.410
For example, these applications pair the cloud checks

343
00:17:44.410 --> 00:17:48.040
to each and every node of the system to ensure that

344
00:17:48.040 --> 00:17:49.690
they're all healthy.

345
00:17:49.690 --> 00:17:52.190
If there's a bad node being detected,

346
00:17:52.190 --> 00:17:54.270
they would replace the node.

347
00:17:54.270 --> 00:17:57.150
Well, it sounds straightforward, right?

348
00:17:57.150 --> 00:18:00.430
Well the part that's interesting about these systems

349
00:18:00.430 --> 00:18:03.880
is that they are very, very powerful.

350
00:18:03.880 --> 00:18:07.480

Replacing a node is a powerful action.

351
00:18:07.480 --> 00:18:11.070
Imagine that Health Guardian determined that half of the

352
00:18:11.070 --> 00:18:13.290
fleet is unhealthy.

353
00:18:13.290 --> 00:18:17.140
Should it go ahead and just replace all those nodes?

354
00:18:17.140 --> 00:18:19.880
Maybe, but usually not.

355
00:18:19.880 --> 00:18:22.723
It really depends on what the problem is.

356
00:18:23.840 --> 00:18:28.360
We built to not have such large failures anyways, right?

357
00:18:28.360 --> 00:18:33.357
So to avoid these automations to take

358
00:18:34.950 --> 00:18:38.973
significant actions that will lead to significant changes,

359
00:18:40.060 --> 00:18:44.020
we installed the bullet counters in place

360
00:18:44.020 --> 00:18:47.963
so that they don't act on (indistinct) signals incorrectly.

361
00:18:49.500 --> 00:18:53.810
So bullet counter in this example would be the one that we

362
00:18:53.810 --> 00:18:56.970
say, "What's the maximum percentage of the fleet

363
00:18:56.970 --> 00:19:00.287
that could be replaced safely at a given time?"

364
00:19:02.120 --> 00:19:06.250
The bullet counters monitor the actions taken

365
00:19:06.250 --> 00:19:11.250
by the Health Guardian, and whenever there is an event where

366
00:19:12.457 --> 00:19:15.200
there are nodes that are being determined that's more than

367
00:19:15.200 --> 00:19:19.960
predefined maximum limit, they will stop the execution,

368
00:19:19.960 --> 00:19:24.520
and instead they will notify the operators to show up

369
00:19:24.520 --> 00:19:26.143
and assess the situation.

370
00:19:28.060 --> 00:19:31.330
Using bullet counters, we ensure that automation

371
00:19:31.330 --> 00:19:34.733
always operates within known and safe limits.

372
00:19:38.040 --> 00:19:40.283
With that, I will hand it over to Colm.

373
00:19:41.480 --> 00:19:42.543
<v ->Thanks, Yasemin.</v>

374
00:19:43.790 --> 00:19:46.180
So as I mentioned briefly earlier,

375
00:19:46.180 --> 00:19:49.090

testing is of enormous importance,

376
00:19:49.090 --> 00:19:52.620
and we invest a lot in testing.

377
00:19:52.620 --> 00:19:56.460
In fact, for our, our most highly available systems,

378
00:19:56.460 --> 00:19:59.360
it's not unusual to spend, you know,

379
00:19:59.360 --> 00:20:04.360
much more time on testing code than writing the code itself.

380
00:20:05.000 --> 00:20:09.390
You know, it's, we have unit tests,

381
00:20:09.390 --> 00:20:12.700
we've got end-to-end tests, we've got integration tests,

382
00:20:12.700 --> 00:20:14.773
and we've even got formal verification.

383
00:20:15.690 --> 00:20:19.643
It's, we've got, you know, pre-production environments.

384
00:20:20.570 --> 00:20:25.470
We, even these days, you know, we'll test how a system

385
00:20:25.470 --> 00:20:28.540
copes with rolling forward to a deployment process,

386
00:20:28.540 --> 00:20:30.950
as well as rolling back through a deployment process

387
00:20:30.950 --> 00:20:33.150
before we ever really deploy to production,

388
00:20:33.150 --> 00:20:34.630
just to make sure that like, well,

389
00:20:34.630 --> 00:20:38.083
if we had to do those things, we would be able to.

390
00:20:39.110 --> 00:20:42.234
If you want to see examples of,

391
00:20:42.234 --> 00:20:44.253
of how we raise the bar on testing,

392
00:20:45.188 --> 00:20:47.970
you can look at our open source projects on GitHub.

393
00:20:47.970 --> 00:20:52.970
So our s2n project, which is our open source SSL and TLS

394
00:20:53.220 --> 00:20:58.220
library is developed by a team at Amazon who work just like

395
00:20:58.480 --> 00:21:01.160
any other team in Amazon, and it's a really simple,

396
00:21:01.160 --> 00:21:05.068
like open view into, into how we work.

397
00:21:05.068 --> 00:21:07.748
And you can see just the sheer staggering number of tests

398
00:21:07.748 --> 00:21:12.100
that are there, that are running on every single build.

399
00:21:12.100 --> 00:21:14.530
You know, when we don't, we don't just run those tests

400
00:21:14.530 --> 00:21:19.530

before we deploy, they're integrated into our CICD process,

401
00:21:20.140 --> 00:21:23.940
and, you know, every time we check in, we run them all,

402
00:21:23.940 --> 00:21:26.323
and we make sure that they all pass before,

403
00:21:27.180 --> 00:21:28.720
before promoting, just to make sure

404
00:21:28.720 --> 00:21:29.880
that there are no regressions.

405
00:21:29.880 --> 00:21:32.140
And that's very, very typical for these

406
00:21:32.140 --> 00:21:34.390
high availability systems.

407
00:21:34.390 --> 00:21:37.417
But, we've been going further.

408
00:21:37.417 --> 00:21:40.860
An so, in the last few years, we've actually been going

409
00:21:40.860 --> 00:21:43.070
further and further with automated reasoning

410
00:21:43.070 --> 00:21:45.060
and formal verification.

411
00:21:45.060 --> 00:21:48.060
So this is something we've been doing for a long time,

412
00:21:48.060 --> 00:21:51.840
you know, on the s2n project, we've been formally verifying

413
00:21:51.840 --> 00:21:56.833
the correctness of parts of s2n for about six years now.

414
00:21:57.800 --> 00:22:00.450
But in, in the last few years,

415
00:22:00.450 --> 00:22:03.293
we've been able to get to the point where now, you know,

416
00:22:05.660 --> 00:22:07.950
I'm gonna say regular developers who don't have

417
00:22:07.950 --> 00:22:10.320
specialized training in automated reasoning

418
00:22:10.320 --> 00:22:13.870
or formal verification are able to use these techniques

419
00:22:13.870 --> 00:22:15.690
because we've improved the tooling,

420
00:22:15.690 --> 00:22:17.343
it's getting easier and easier.

421
00:22:18.350 --> 00:22:20.760
And this is amazing because formal verification, you know,

422
00:22:20.760 --> 00:22:24.670
tests can always prove that code is correct for a particular

423
00:22:24.670 --> 00:22:27.440
input, but formal verification and can prove that that code

424
00:22:27.440 --> 00:22:29.290
is correct for any particular input.

425
00:22:29.290 --> 00:22:31.310

So it's very, very powerful.

426
00:22:31.310 --> 00:22:35.090
We can find, you know, really hard to find edge cases

427
00:22:35.090 --> 00:22:37.490
that you won't find using any other technique.

428
00:22:37.490 --> 00:22:39.310
And it's pretty awesome now that, you know,

429
00:22:39.310 --> 00:22:43.290
we're able to get to the point where, you know,

430
00:22:43.290 --> 00:22:45.540
regular software developers like me can,

431
00:22:45.540 --> 00:22:49.980
can actually use these tools and improve things

432
00:22:49.980 --> 00:22:50.813
about their code.

433
00:22:50.813 --> 00:22:51.900
And if you're interested in some of that,

434
00:22:51.900 --> 00:22:54.340
I'd encourage you to check out CBMC,

435
00:22:54.340 --> 00:22:58.310
which is a tool we've been using on s2n that is very

436
00:22:58.310 --> 00:23:03.310
intuitive, and I found very easy to use for developers.

437
00:23:03.400 --> 00:23:08.020
So, the next item on our list is something called

438
00:23:08.020 --> 00:23:12.720
lifecycle management, which we found that, you know,

439
00:23:12.720 --> 00:23:16.350
to get high availability over long periods of time

440
00:23:16.350 --> 00:23:19.820
in our systems, we need to be very, very intentional about

441
00:23:19.820 --> 00:23:23.620
how we manage the life cycle of a lot of aspects

442
00:23:23.620 --> 00:23:24.623
of those systems.

443
00:23:25.853 --> 00:23:29.730
And in particular, any kind of credentials that are used

444
00:23:29.730 --> 00:23:30.563
by those systems.

445
00:23:30.563 --> 00:23:33.920
So that's things like keys and certificates and so on.

446
00:23:33.920 --> 00:23:36.700
So modern security and compliance frameworks demand that

447
00:23:36.700 --> 00:23:39.000
credentials be frequently rotated, and,

448
00:23:39.000 --> 00:23:41.780
and that makes sense, no one wants a key or a certificate

449
00:23:41.780 --> 00:23:44.580
around that could be used for very long periods of time.

450
00:23:45.590 --> 00:23:48.700

But at the same time, expired and mismatched credentials

451
00:23:48.700 --> 00:23:50.430
could be a source of outages, right?

452
00:23:50.430 --> 00:23:52.770
If a certificate or key expires and it's still

453
00:23:52.770 --> 00:23:55.260
in your system, that's not gonna be good.

454
00:23:55.260 --> 00:23:58.920
So we've learned to decouple that expiry from alarming.

455
00:23:58.920 --> 00:24:02.930
So we, we alarm well before the expiry time of any key

456
00:24:02.930 --> 00:24:07.000
or credential, and we've learned to be super intentional

457
00:24:07.000 --> 00:24:11.800
and kind of go into overkill on how to monitor it.

458
00:24:11.800 --> 00:24:14.530
So we've got time to expiry metrics for anything

459
00:24:14.530 --> 00:24:15.800
that expires.

460
00:24:15.800 --> 00:24:18.700
We look at that from both the server side perspective,

461
00:24:18.700 --> 00:24:20.620
so the thing that might be serving a certificate

462
00:24:20.620 --> 00:24:23.090
or using a key, and the client side,

463
00:24:23.090 --> 00:24:25.083
thing that's connecting to it.

464
00:24:25.083 --> 00:24:27.540
Like I said, we alarm and investigate well before

465
00:24:27.540 --> 00:24:29.220
there's any kind of a problem.

466
00:24:29.220 --> 00:24:32.500
And, and then on top of that, we've got additional

467
00:24:32.500 --> 00:24:35.340
fail safes, and canaries that are constantly scanning

468
00:24:35.340 --> 00:24:38.370
for anything that looks like it's even close to expiry,

469
00:24:38.370 --> 00:24:41.080
so that we've got another safety net there too.

470
00:24:41.080 --> 00:24:43.760
And then, with all of those systems combined,

471
00:24:43.760 --> 00:24:45.090
we've learned that, you know,

472
00:24:45.090 --> 00:24:48.650
that means we need to deploy any new key or credential,

473
00:24:48.650 --> 00:24:51.030
make sure it's absolutely everywhere that could need it

474
00:24:51.030 --> 00:24:53.787
before we activate it, and only then activate it

475
00:24:53.787 --> 00:24:55.820

and kind of do the same in reverse whenever

476
00:24:55.820 --> 00:24:59.290
we're de deactivating or evoking a key or certificate.

477
00:24:59.290 --> 00:25:00.830
And we found that, you know,

478
00:25:00.830 --> 00:25:03.810
paying particular attention to detail on those processes

479
00:25:03.810 --> 00:25:07.660
has been really key to avoiding just any kind of outage

480
00:25:07.660 --> 00:25:11.630
you could see from just expired or mismatched keys

481
00:25:11.630 --> 00:25:13.200
and credentials.

482
00:25:13.200 --> 00:25:16.210
And so with that, I'll hand over to Yasemin,

483
00:25:16.210 --> 00:25:19.050
who's gonna tell us about modular separations.

484
00:25:19.050 --> 00:25:20.207
<v ->Thank you, Colm.</v>

485
00:25:22.036 --> 00:25:23.253
Modular separation.

486
00:25:24.270 --> 00:25:27.300
We avoid multi-link architectures and decouple

487
00:25:27.300 --> 00:25:29.724
individual responsibilities into their own

488
00:25:29.724 --> 00:25:31.474
dedicated components.

489
00:25:33.480 --> 00:25:35.870
Control plane versus Data plane separation

490
00:25:35.870 --> 00:25:37.800
is a good example of this.

491
00:25:37.800 --> 00:25:42.027
Most data services have this notion of control plane

492
00:25:42.027 --> 00:25:44.100
and data plane APIs.

493
00:25:44.100 --> 00:25:46.733
Let's take Kinesis data streams as an example.

494
00:25:47.800 --> 00:25:51.270
Kinesis data streams is a real time streaming service

495
00:25:51.270 --> 00:25:54.780
that enables customers to write records into a log stream,

496
00:25:54.780 --> 00:25:55.973
and read them later on.

497
00:25:57.044 --> 00:25:58.820
And the very first thing.

498
00:25:58.820 --> 00:26:02.070
The very first thing customers do when they start using

499
00:26:02.070 --> 00:26:05.520
the service, they go ahead and create a stream.

500
00:26:05.520 --> 00:26:08.147

Create stream is a control plane API.

501
00:26:09.003 --> 00:26:10.700
Once the stream is created,

502
00:26:10.700 --> 00:26:13.470
producer applications can start ingesting the data

503
00:26:13.470 --> 00:26:17.300
continuously and consumer applications read those records

504
00:26:17.300 --> 00:26:18.453
within milliseconds.

505
00:26:19.960 --> 00:26:23.716
The APIs used by the producer and consumer applications

506
00:26:23.716 --> 00:26:25.293
are the data plane APIs.

507
00:26:26.396 --> 00:26:27.696
As we see in this example,

508
00:26:28.687 --> 00:26:33.687
access patterns of these two, these two type of APIs

509
00:26:34.270 --> 00:26:35.103
are different.

510
00:26:36.470 --> 00:26:40.500
Consequently, their dependencies are very different as well.

511
00:26:40.500 --> 00:26:44.900
For example, control plane APIs are depending on the

512
00:26:44.900 --> 00:26:49.900
asynchronous workflows to execute the steps of creating

513
00:26:50.390 --> 00:26:55.390
a stream, versus data plane APIs are dependent on the data

514
00:26:55.804 --> 00:26:58.433
store to fetch the records within milliseconds.

515
00:26:59.970 --> 00:27:04.000
By decoupling the two API types, we limit the blast,

516
00:27:04.000 --> 00:27:07.970
we limit the impact their dependencies could be creating.

517
00:27:07.970 --> 00:27:10.950
For example, if there's an event that's going on

518
00:27:10.950 --> 00:27:14.170
in the Async workflows, then it's going to be only

519
00:27:14.170 --> 00:27:16.850
the control plane API's that are being impacted

520
00:27:16.850 --> 00:27:19.713
while data plane APIs continue to work.

521
00:27:23.550 --> 00:27:26.260
Next technique is static stability.

522
00:27:26.260 --> 00:27:28.540
And this technique works hand to hand

523
00:27:28.540 --> 00:27:29.740
with modular separation.

524
00:27:32.020 --> 00:27:35.830
Availability of a system can be as good as

525
00:27:35.830 --> 00:27:37.853

availability of its dependencies.

526
00:27:38.830 --> 00:27:41.820
For this reason, we strive to keep the dependencies of

527
00:27:41.820 --> 00:27:43.663
systems to absolute minimum.

528
00:27:46.520 --> 00:27:49.400
Let's get back to control plane versus data plane

529
00:27:49.400 --> 00:27:50.363
discussion again.

530
00:27:51.710 --> 00:27:54.400
The dependency here that I didn't discuss before

531
00:27:54.400 --> 00:27:56.190
is the Metadata store.

532
00:27:56.190 --> 00:27:59.989
Metadata store is the one that process the information

533
00:27:59.989 --> 00:28:03.023
about resources, Kinesis streams in this case.

534
00:28:04.070 --> 00:28:08.200
So stream creation execution has direct dependency

535
00:28:08.200 --> 00:28:11.240
to the availability of Metadata store because it needs

536
00:28:11.240 --> 00:28:14.700
to process the information that the stream is created.

537
00:28:14.700 --> 00:28:18.463
Imagine there's an outage on Metadata store.

538
00:28:18.463 --> 00:28:20.750
We expect the Async workflows to be impacted

539
00:28:20.750 --> 00:28:23.230
because it cannot access the data store anymore,

540
00:28:23.230 --> 00:28:26.280
therefore the control plane APIs are impacted.

541
00:28:26.280 --> 00:28:30.500
Let's check the impact of Metadata store on data plane APIs.

542
00:28:31.720 --> 00:28:36.720
Well, data plane APIs also need to know about these streams,

543
00:28:36.977 --> 00:28:39.590
and where they are located so that they can serve,

544
00:28:39.590 --> 00:28:41.383
put and get APIs on them.

545
00:28:42.360 --> 00:28:45.010
So if the data plane APIs have direct and synchronous

546
00:28:45.010 --> 00:28:49.690
dependency on Metadata store, then its outage

547
00:28:49.690 --> 00:28:51.603
will also impact the data plane APIs.

548
00:28:53.300 --> 00:28:54.910
Well, that's not great.

549
00:28:54.910 --> 00:28:58.120
It's large blast radius impact.

550
00:28:58.120 --> 00:29:01.250

So, the sole dependency of data plane APIs

551
00:29:01.250 --> 00:29:03.650
is really the data store, they are responsible

552
00:29:03.650 --> 00:29:06.660
for serving customer records.

553
00:29:06.660 --> 00:29:10.610
So, considering the static stability principle,

554
00:29:10.610 --> 00:29:14.970
let's look at this design and see how we can eliminate

555
00:29:14.970 --> 00:29:18.623
the Metadata store dependency from data plane APIs.

556
00:29:19.940 --> 00:29:24.923
We eliminate this dependency by moving the Metadata store

557
00:29:24.923 --> 00:29:29.818
dependency of data plane from being a synchronous dependency

558
00:29:29.818 --> 00:29:31.793
to being an asynchronous dependency.

559
00:29:33.010 --> 00:29:38.010
In this architecture, the Metadata store keeps track

560
00:29:38.170 --> 00:29:40.293
of the stream of information.

561
00:29:41.190 --> 00:29:46.040
And here it has a copy of this information

562
00:29:46.040 --> 00:29:48.183
stored within the data plane itself.

563
00:29:51.420 --> 00:29:54.667
Any updates that are being applied to this metadata

564
00:29:54.667 --> 00:29:56.920
gets propagated to the data plane,

565
00:29:56.920 --> 00:29:59.653
and it's own snapshot asynchronously.

566
00:30:01.190 --> 00:30:04.770
In this case, if there's an outage in Metadata store,

567
00:30:04.770 --> 00:30:08.240
it's the control plane APIs those will be impacted,

568
00:30:08.240 --> 00:30:11.520
but the data plane APIs will continue to work

569
00:30:11.520 --> 00:30:15.790
by using the smaller version of their snapshots

570
00:30:15.790 --> 00:30:17.893
that are stored within themselves.

571
00:30:19.660 --> 00:30:23.980
This way, we ensure that the data plane is statically stable

572
00:30:23.980 --> 00:30:27.733
by using its absolute minimum set of dependencies.

573
00:30:28.730 --> 00:30:29.760
<v ->Thanks Yasemin.</v>

574
00:30:31.390 --> 00:30:35.640
So our ninth and penultimate item is constant work,

575
00:30:35.640 --> 00:30:36.473

which is,

576
00:30:37.920 --> 00:30:40.210
you know, whenever we were learning big O notation,

577
00:30:40.210 --> 00:30:42.410
if you study computer science, you know,

578
00:30:42.410 --> 00:30:44.610
you probably learnt about systems that are constant of work,

579
00:30:44.610 --> 00:30:46.670
that means 0 1, right?

580
00:30:46.670 --> 00:30:48.570
A big O notation.

581
00:30:48.570 --> 00:30:50.710
And that turns out to be a very important concept

582
00:30:50.710 --> 00:30:53.860
for some of our highly available systems.

583
00:30:53.860 --> 00:30:56.990
You know, in general, right?

584
00:30:56.990 --> 00:31:00.970
Risk is proportionate to rates of change in systems, right?

585
00:31:00.970 --> 00:31:03.680
A spike in load, for example, right?

586
00:31:03.680 --> 00:31:05.710
Can cause the system to slow down, right?

587
00:31:05.710 --> 00:31:09.070
And then the system gets into a mode that it's not used to

588
00:31:09.070 --> 00:31:11.560
operating in, things don't really know how to handle that,

589
00:31:11.560 --> 00:31:13.630
they might start timing out and so on.

590
00:31:13.630 --> 00:31:16.550
And the, you know, issues like that can cause

591
00:31:16.550 --> 00:31:18.380
cascading failures.

592
00:31:18.380 --> 00:31:21.890
And so we've learned that reducing the overall dynamism

593
00:31:21.890 --> 00:31:24.090
in the system, you know, the amount of change it can,

594
00:31:24.090 --> 00:31:25.530
it can even go through,

595
00:31:25.530 --> 00:31:28.110
is a useful way to make them simpler and to reduce

596
00:31:28.110 --> 00:31:29.230
all that risk.

597
00:31:29.230 --> 00:31:31.810
And a kind of counter-intuitive solution to kind of wrangle

598
00:31:31.810 --> 00:31:35.060
that risk, is actually to run the system at maximum load

599
00:31:35.060 --> 00:31:36.410
all the time, right?

600
00:31:36.410 --> 00:31:38.170

And even though that sounds like, "Well, now the system's

601
00:31:38.170 --> 00:31:40.250
gonna be maxed out all the time."

602
00:31:40.250 --> 00:31:42.850
It actually reduces the amount of dynamism and change

603
00:31:42.850 --> 00:31:45.200
that's in the system, and therefore risk.

604
00:31:45.200 --> 00:31:48.210
A really simple example of that is, how we apply

605
00:31:48.210 --> 00:31:53.000
this constant work pattern to say configuration changes.

606
00:31:53.000 --> 00:31:55.940
So a really common pattern and how developers manage

607
00:31:57.070 --> 00:31:58.800
configuration changes is like, you know,

608
00:31:58.800 --> 00:32:01.770
customer makes a change, and that change gets ingested

609
00:32:01.770 --> 00:32:04.870
into the system as like a Delta, you know, do this thing,

610
00:32:04.870 --> 00:32:07.777
do X, do Y, and that goes into a workflow,

611
00:32:07.777 --> 00:32:10.730
and the workflow manages getting that change out to all of

612
00:32:10.730 --> 00:32:13.320
the systems that need to reflect that change.

613
00:32:13.320 --> 00:32:16.660
That works, that's a, a simple pattern,

614
00:32:16.660 --> 00:32:18.670
but the problem is, when lots of changes happen,

615
00:32:18.670 --> 00:32:20.480
so lots of customers at the same time,

616
00:32:20.480 --> 00:32:22.450
maybe it's a particularly busy day or whatever,

617
00:32:22.450 --> 00:32:24.220
the overall system slows down, right?

618
00:32:24.220 --> 00:32:27.730
Because the workflow's got more work to do, right?

619
00:32:27.730 --> 00:32:31.420
And as I said, it could, that can, you know, hit you in,

620
00:32:31.420 --> 00:32:33.420
in ways that cascade.

621
00:32:33.420 --> 00:32:35.430
So a simpler version of this is imagine,

622
00:32:35.430 --> 00:32:37.680
well every customer, they make their change,

623
00:32:37.680 --> 00:32:42.140
and it's just reflected as say, a file or a key in S3,

624
00:32:42.140 --> 00:32:43.140
that's it, right?

625
00:32:43.140 --> 00:32:46.470

They just make their change and it's effectively a file

626
00:32:46.470 --> 00:32:48.460
or a key in S3.

627
00:32:48.460 --> 00:32:51.310
And the system on the other side, instead of using

628
00:32:51.310 --> 00:32:54.470
a workflow, all it's doing is checking all of those files

629
00:32:54.470 --> 00:32:55.490
every single time.

630
00:32:55.490 --> 00:32:58.047
Just downloading every single file from S3

631
00:32:58.047 --> 00:33:00.670
and using that as its configuration.

632
00:33:00.670 --> 00:33:03.190
And so, even when lots and lots of customers make changes

633
00:33:03.190 --> 00:33:06.530
at the same time, so maybe a hundred files changed.

634
00:33:06.530 --> 00:33:08.900
If the system was always pulling, you know,

635
00:33:08.900 --> 00:33:11.600
all 100 files or all thousand files,

636
00:33:11.600 --> 00:33:13.580
if there's a thousand customers, and just doing that

637
00:33:13.580 --> 00:33:17.010
as its configuration, there's no change, or difference,

638
00:33:17.010 --> 00:33:20.800
or dynamism on the right-hand side of these diagrams,

639
00:33:20.800 --> 00:33:22.530
which reduces the overall risk,

640
00:33:22.530 --> 00:33:24.610
and it could be enormously effective.

641
00:33:24.610 --> 00:33:26.020
And we've learned to apply this pattern

642
00:33:26.020 --> 00:33:28.220
in some really key places.

643
00:33:28.220 --> 00:33:31.170
We've got a builder's library article where we talk about

644
00:33:31.170 --> 00:33:33.260
how we apply this in our health check systems,

645
00:33:33.260 --> 00:33:36.000
in our DNS fail-over systems, so that they can be

646
00:33:36.000 --> 00:33:38.210
incredibly reliable.

647
00:33:38.210 --> 00:33:40.730
And I found it really, really useful.

648
00:33:40.730 --> 00:33:44.560
And with that, Yasemin's gonna take over and close us out

649
00:33:44.560 --> 00:33:47.623
and, and tell us our final lesson.

650
00:33:48.751 --> 00:33:49.950

<v ->Retries.</v>

651
00:33:49.950 --> 00:33:51.830
Retries are somewhat well-known, right?

652
00:33:51.830 --> 00:33:56.210
When there is a failure, we retry and that helps

653
00:33:56.210 --> 00:33:57.950
resolving the problem.

654
00:33:57.950 --> 00:33:59.453
But do they always help?

655
00:34:00.300 --> 00:34:02.420
Let's dig in a little bit.

656
00:34:02.420 --> 00:34:05.370
The problem with retries is that when they are not

657
00:34:05.370 --> 00:34:09.060
used properly, they can cause a larger event

658
00:34:09.060 --> 00:34:12.110
compared to what they're trying to mitigate.

659
00:34:12.110 --> 00:34:14.820
We call this thundering herd problem.

660
00:34:14.820 --> 00:34:17.071
I'll first explain what this problem is,

661
00:34:17.071 --> 00:34:21.870
and then I'll discuss two techniques to avoid it.

662
00:34:21.870 --> 00:34:24.160
Imagine there are transient failures going on

663
00:34:24.160 --> 00:34:25.053
in the system.

664
00:34:26.050 --> 00:34:28.320
Clients start to retry.

665
00:34:28.320 --> 00:34:31.470
As clients retry more, there's more traffic being

666
00:34:31.470 --> 00:34:35.150
generated, so system gets overloaded, and when systems get

667
00:34:35.150 --> 00:34:38.730
overloaded, we discussed, they start load shedding,

668
00:34:38.730 --> 00:34:41.310
and there are more failures, and there are more retries,

669
00:34:41.310 --> 00:34:45.093
and this is, it's this vicious cycle that's keep going.

670
00:34:46.120 --> 00:34:48.940
Creating more work in the system with retries

671
00:34:48.940 --> 00:34:51.570
doesn't really have to solve the problem.

672
00:34:51.570 --> 00:34:54.203
So how do we avoid the thundering herd issue?

673
00:34:55.170 --> 00:34:57.270
The first technique I'll discuss

674
00:34:57.270 --> 00:35:00.640
is exponential backoff and jitter.

675
00:35:00.640 --> 00:35:04.650

When clients retry without exponential backoff and jitter,

676
00:35:04.650 --> 00:35:08.640
the same amount of traffic hits the service right around

677
00:35:08.640 --> 00:35:12.610
the same time with the same frequency.

678
00:35:12.610 --> 00:35:17.480
In this example, there's a second delay between each retry,

679
00:35:17.480 --> 00:35:21.403
and the same set of calls are being hit.

680
00:35:22.260 --> 00:35:24.493
Calls are used to hit the service.

681
00:35:25.830 --> 00:35:28.380
So I have two clients represented here

682
00:35:28.380 --> 00:35:32.530
with the shades of colors, and first line is doing one TPS,

683
00:35:32.530 --> 00:35:35.860
second one is doing three, and the third one is doing five

684
00:35:35.860 --> 00:35:38.113
TPS transactions per second, right?

685
00:35:39.750 --> 00:35:42.423
They're hitting service again and again.

686
00:35:43.660 --> 00:35:48.660
So, this doesn't help to lower the load on the service.

687
00:35:49.860 --> 00:35:53.146
With exponential backoff and jitter being used,

688
00:35:53.146 --> 00:35:54.750
we apply two techniques.

689
00:35:54.750 --> 00:35:59.750
The first one is, we give wait time between each retry

690
00:36:01.380 --> 00:36:04.450
and that wait time increases exponentially

691
00:36:04.450 --> 00:36:06.400
in between every retries.

692
00:36:06.400 --> 00:36:09.090
And the second aspect of this is jitter.

693
00:36:09.090 --> 00:36:12.820
We add randomness so that when the retry comes on,

694
00:36:12.820 --> 00:36:17.820
it will come in on that, the retries will not come in

695
00:36:18.130 --> 00:36:21.403
right at the same second, but rather it will be distributed

696
00:36:21.403 --> 00:36:23.023
across the timeframe.

697
00:36:24.630 --> 00:36:26.330
This technique helps,

698
00:36:26.330 --> 00:36:29.223
but we find it's not always sufficient at scale.

699
00:36:33.090 --> 00:36:35.210
Let's look at client throttling.

700
00:36:35.210 --> 00:36:37.130

We found client throttling to be much more

701
00:36:37.130 --> 00:36:38.423
effective technique.

702
00:36:39.950 --> 00:36:43.150
AWS the case have built in support for client throttling

703
00:36:43.150 --> 00:36:44.470
as well.

704
00:36:44.470 --> 00:36:47.860
This technique keeps local state on the client

705
00:36:47.860 --> 00:36:52.270
and decides to retry or not according to the local state

706
00:36:52.270 --> 00:36:54.210
that it's keeping track of.

707
00:36:54.210 --> 00:36:56.260
The talking back targeting is being used

708
00:36:56.260 --> 00:36:58.220
to download this property.

709
00:36:58.220 --> 00:37:01.050
Let's say we have a client that's making a thousand

710
00:37:01.050 --> 00:37:04.630
requests per second, that's the purple line that I have

711
00:37:04.630 --> 00:37:06.217
on the graph.

712
00:37:06.217 --> 00:37:07.860
And it's static, it's not changing,

713
00:37:07.860 --> 00:37:10.850
always making a thousand requests per second.

714
00:37:10.850 --> 00:37:14.900
And the red line on the graph is the failure rate.

715
00:37:14.900 --> 00:37:18.080
Let's imagine that the service starting to have failure,

716
00:37:18.080 --> 00:37:21.357
it's initially it's healthy, it runs tests,

717
00:37:21.357 --> 00:37:24.083
having more and more failures, by the middle of the graph

718
00:37:24.083 --> 00:37:27.030
there's a hundred percent failures going on in the system.

719
00:37:27.030 --> 00:37:30.940
And then the failure start to get better over time.

720
00:37:30.940 --> 00:37:31.793
Back to zero.

721
00:37:32.800 --> 00:37:35.710
So let's look at the system when there is no

722
00:37:35.710 --> 00:37:37.990
client throttling is used.

723
00:37:37.990 --> 00:37:42.660
What happens is, the client starts retrying

724
00:37:42.660 --> 00:37:46.650
as it starts to observe the failures, and that excess

725
00:37:46.650 --> 00:37:49.330

traffic that's being created by the clients

726
00:37:49.330 --> 00:37:53.540
is following the same shape

727
00:37:53.540 --> 00:37:56.650
that the failure graph that we were just looking at, right?

728
00:37:56.650 --> 00:37:58.940
As there are more failures, there is more work

729
00:37:58.940 --> 00:38:00.440
being created.

730
00:38:00.440 --> 00:38:02.300
And towards the end off the middle of the graph,

731
00:38:02.300 --> 00:38:06.620
as the service is at hundred percent, let's say,

732
00:38:06.620 --> 00:38:11.560
the failure rate is the max, the retry count is the maximum,

733
00:38:11.560 --> 00:38:14.600
is the maximum amount of traffic being generated,

734
00:38:14.600 --> 00:38:16.483
and none of that being served.

735
00:38:17.410 --> 00:38:22.010
So, retrying in this case is not helping to resolve

736
00:38:22.010 --> 00:38:23.600
the issue, right?

737
00:38:23.600 --> 00:38:27.313
So let's see how client throttling helps this problem.

738
00:38:29.380 --> 00:38:32.223
So when the client throttling is enabled,

739
00:38:33.470 --> 00:38:36.840
as the service starts having more and more failures,

740
00:38:36.840 --> 00:38:39.797
the clients actually recognize it,

741
00:38:39.797 --> 00:38:43.930
and starts lowering the retry rate on the client side,

742
00:38:43.930 --> 00:38:46.310
and by the time service is having a hundred percent

743
00:38:46.310 --> 00:38:48.190
failing in the middle of the graph,

744
00:38:48.190 --> 00:38:49.840
there are no retries going on,

745
00:38:49.840 --> 00:38:51.980
it's just a flat number of requests

746
00:38:51.980 --> 00:38:54.890
that are still being sent, and when failure is seen,

747
00:38:54.890 --> 00:38:58.840
no retry, because it knows that retrying will not help.

748
00:38:58.840 --> 00:39:01.820
But as soon as service starts recovering in the second half

749
00:39:01.820 --> 00:39:06.070
of the graph, the retries start to pick up again,

750
00:39:06.070 --> 00:39:09.150

because now it knows that there's a chance a second retry

751
00:39:09.150 --> 00:39:11.693
might actually get a successful response.

752
00:39:13.080 --> 00:39:15.720
We found that this technique is much more effective

753
00:39:15.720 --> 00:39:19.650
to improve available depositure of client applications

754
00:39:19.650 --> 00:39:23.460
while not generating any unnecessary work on the system.

755
00:39:23.460 --> 00:39:27.990
All right, that concludes our ten points today,

756
00:39:27.990 --> 00:39:29.340
and thank you for watching.

757
00:39:30.817 --> 00:39:34.067
(cheerful music plays)

758
00:39:39.664 --> 00:39:43.247
(cheerful music continues)