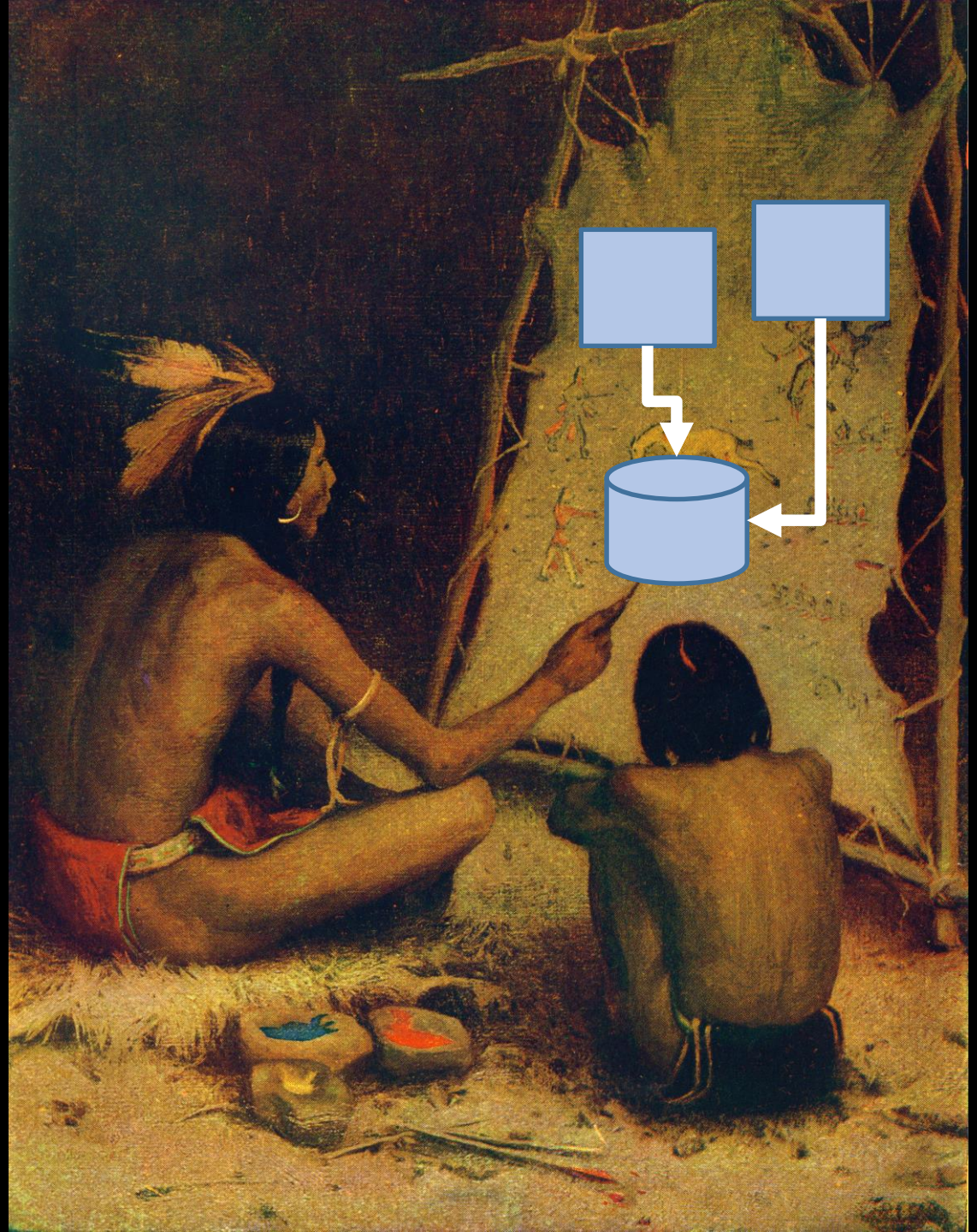
Architecture Decision Records in Action

Michael Keeling
IBM
@michaelkeeling

Joe Runde
IBM
@joerunde

How do you share important design decisions?



Oral history is a great way to share design decisions while you're still exploring the architecture.

Limits of Oral History

Short reach

Time consuming to share

Dies without constant attention

Changes over time

Before long, important details are forgotten.

…unless you write something down.


UP TO DATE DOCUMENTATION
NOW THERE IS SOMETHING I HAVEN'T SEEN IN A LONG TIME.
imgflip.com

What is the least we can document and still remain effective?

Our ideal documentation…

Low barrier to entry

Minimal training

Value-adding, useful

Easy to update

Something skeptics will accept

ARCHITECTURE DECISION RECORDS

“An architecture decision record is a short text file in a format similar to an Alexandrian pattern that describes a set of forces and a single decision in response to those forces.”

Documenting Architecture Decisions by Michael Nygard
http://thinkrelevance.com/blog/2011/11/15/documenting-architecture-decisions

"An architecture decision record is a short text file in a format similar to an Alexandrian pattern that describes a set of forces and a single decision in response to those forces."

Documenting Architecture Decisions by Michael Nygard
http://thinkrelevance.com/blog/2011/11/15/documenting-architecture-decisions

Architecture Decision Record (ADR)

Design decision

Context

Rationale

Implications

Joseph-Runde Address comments — cc6d284 on Dec 12, 2016

1 contributor

52 lines (38 sloc) 2.02 KB — Raw | Blame | History

ADR 5: Use a pipe and filter architecture

Preparing a set of training data for the ranker requires executing many tasks. These tasks need to be organized somehow.

Previously we tried java and ended up with a mess of nested worker pools and confusion about execution and data flow through the process. We want the structure of the project to make the control flow, data flow, and parallelization readily apparent. (See cranberry for the clusterfluff)

We picked `Go` for this project partly because it has flexible high-level functionality, so we want to make good use of some go constructs.

Decision

We will organize the agent with the pipe-and-filter pattern, using go channels and goroutines to create the pipeline.

Rationale

This could probably be done in a single function script in <100 lines, but a clear pattern will give us architecturally evident control and data flow, testability, and modifiability/extensibility. We need these qualities because we have to support and maintain this code.

The pipe and filter pattern fits well with the problem at hand- we have multiple discrete steps operating in a fixed order, which sometimes operate concurrently, and can parallelize work.

There's literally a tutorial on building pipelines with `chan`s on the go blog. TL;DR is that pipe and filter is super easy in go because:

- Go has easy threading, e.g. `go foo()` runs foo in a new thread (goroutine)
- Go has the `chan` construct which safely passes data between threads and allows synchronization

Status

Accepted

Consequences

- We will need to learn about all the quirks of go `chan`s
- Go doesn't have generics, so the generalization of a "step" in a pipeline is impossible to do in a not-stupid way
  - We'll need separate interfaces for each step and separate mocks in all the tests
- Implementing and altering steps without affecting the rest of the workflow will be easy

Let’s see an example!

Plain, direct language

Brief, 1-2 pages max

Markdown

Stored with the code

Our ADR Template

- Number, title
- Context - Value neutral, describe forces at play
- Decision
  - 1 sentence, “We will ….”
- Status
  - Proposed, Accepted, Deprecated, Superseded
- Consequences
  - New context after decision applied
  - Go beyond the obvious

```
# ADR N: Brief Decision Title

Context goes here. Describe the forces at play, including technological, political,
social, and project local. These forces are likely in tension, and should be called
out as such. The language in this section is value-neutral. It is simply describing
facts.  Rationale should be self-evident from the context

## Decision

This section describes our response to these forces. It is stated in full sentences,
with active voice. "We will ...“

## Status

choose one: [Proposed | Accepted | Deprecated | Superseded]

if deprecated, include a rationale. If superseded, include a link to the new ADR

## Consequences

Describe the resulting context, after applying the decision. All consequences should
be listed here, not just the "positive" ones. A particular decision may have positive,
negative, and neutral consequences, but all of them affect the team and project in the
future.
```

readme.md

Architecture Decision Records

What is an Architecturally Significant Decision?

See Documenting Architecture Decisions and Architecture decisions: demystifying architecture for more information.

The basic idea is to capture key decisions related to anything "architectural" in a way that promotes better communication than simple word-of-mouth. ADRs are a part of our overall design communication strategy and are not necessarily the sole source of documenting the architecture. Documenting decisions as they happen is a great way to build up history and communicate latest thinking about the overall architecture design.

Template

Use this template in any new ADRs. Replace the help text as you write the ADR.

```
# ADR N: Brief Decision Title

Context goes here.
```

Record any architecture design decision

- Alters externally visible system properties
- Modifies a public interfaces
- Directly influences a high priority quality attribute
- Includes or removes a dependency
- Direct result of new information about a constraint
- Accepts strategic technical debt
- Changes the general structures of the system
- Forces developers to change their development approach

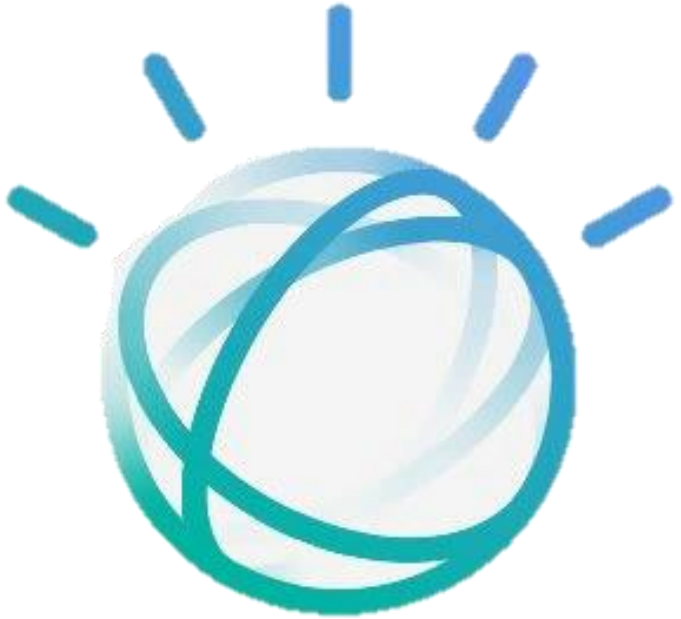
OUR EXPERIENCE WITH ADRS

Experience Context

Team Size	~9 engineers
Team Background	Software engineering Machine learning Computer science
Team Experience	1 – 20+ years mean 5, median 2
Process	Scrum + XP

We help build Watson

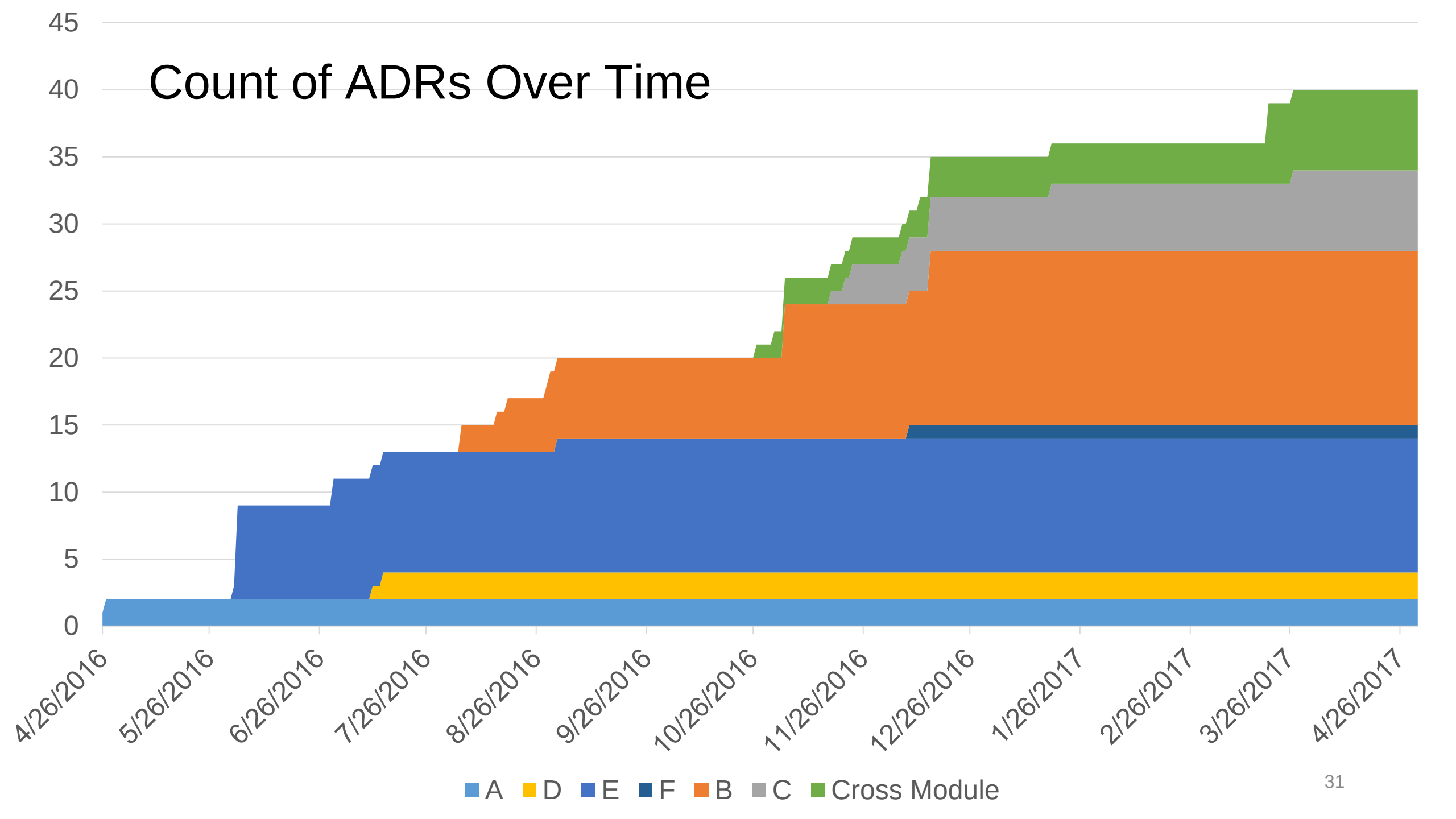
- Cloud-based microservices
- Many products in Watson
- ~5 teams in our Watson neighborhood (product area)
- 25+ microservices in our neighborhood
- Strategic governance across Watson
  - Security
  - Cloud platform
  - Broad architectural patterns

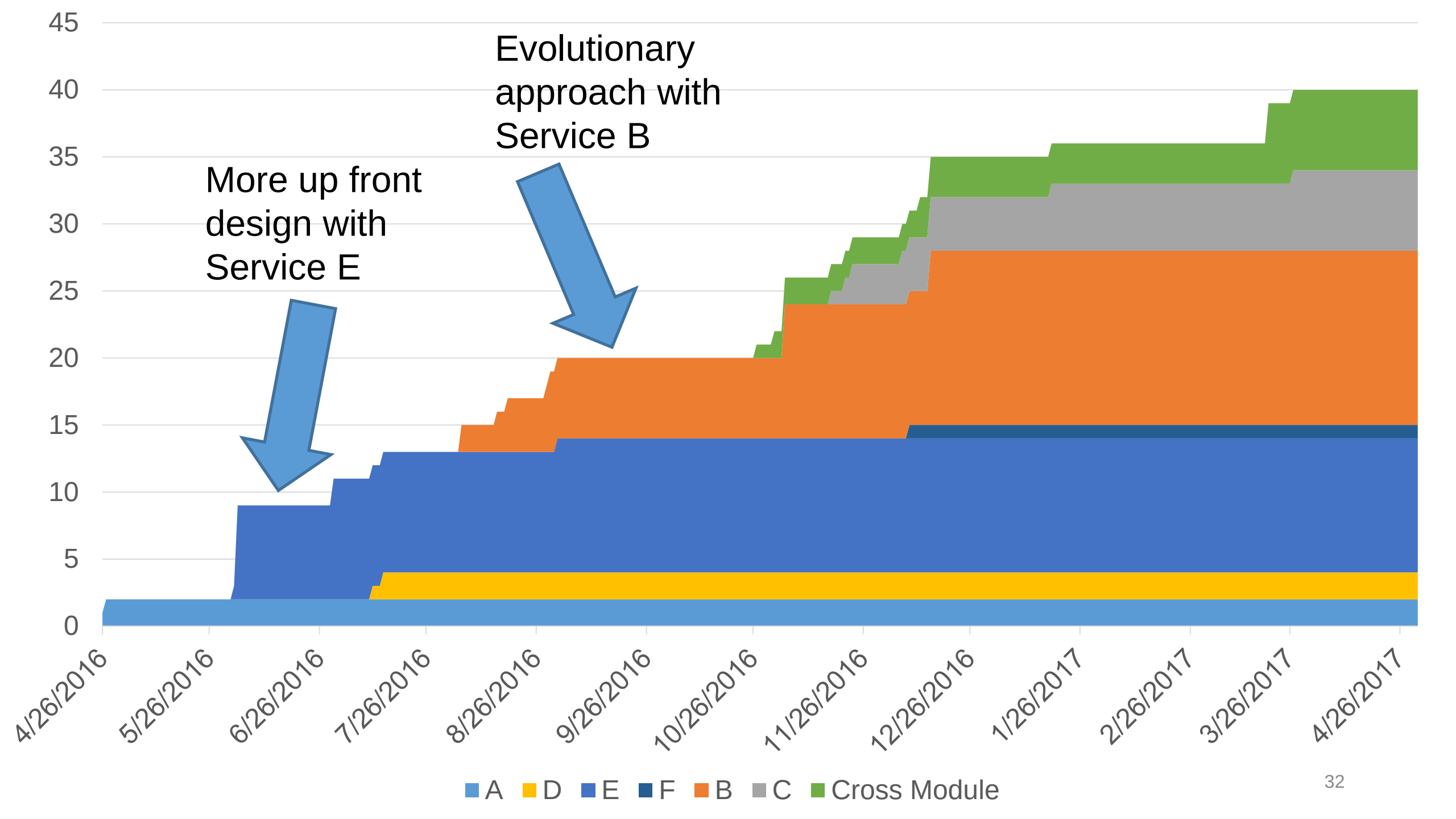


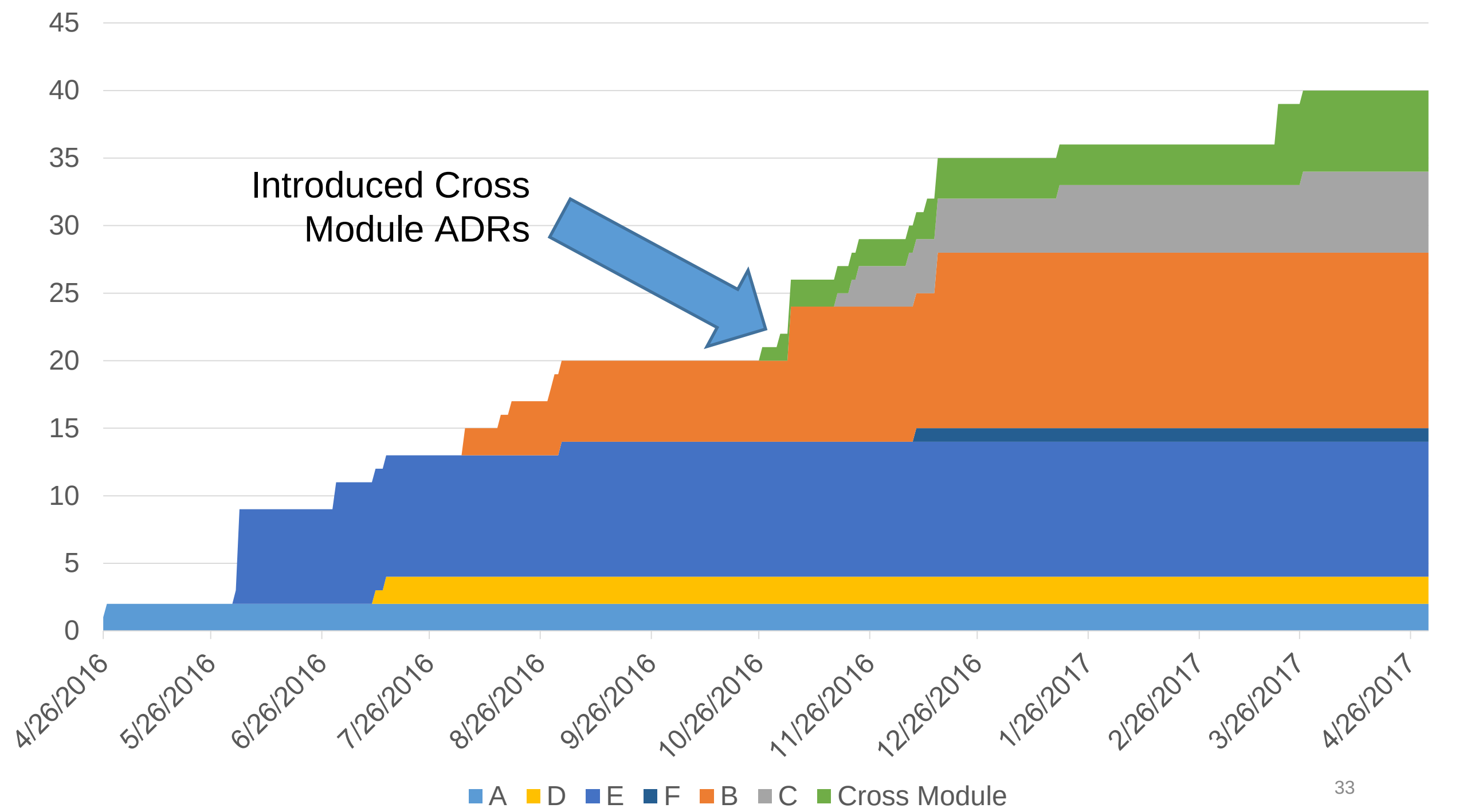
Total ADRs Since April 2016

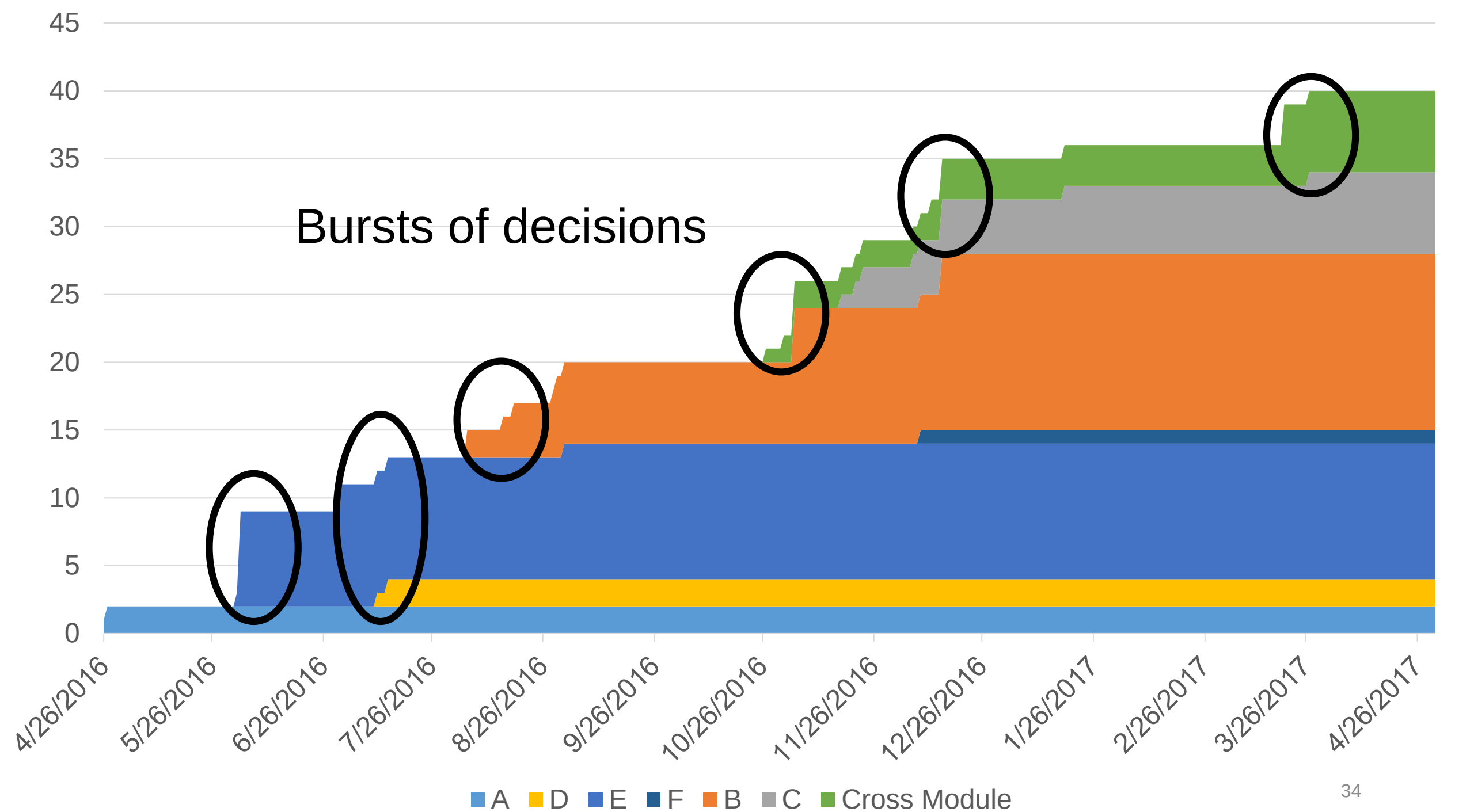
Service / Repo	Count of ADRs	Language	KLOC*
A (killed)	2	Java	-
B	13	Java	11
C	6	Go	5.5
Cross Module	6	Markdown	N/A
D (killed)	2	Java	-
E	10	Java	2.6
F	1	Java	4
G, H, I	0	Go, Java	-
Grand Total	**40**		

*git ls-files | xargs cat | wc -l


Count of ADRs Over Time
45
40
35
30
25
20
15
10
5
0
4/26/2016
5/26/2016
6/26/2016
7/26/2016
8/26/2016
9/26/2016
10/26/2016
11/26/2016
12/26/2016
1/26/2017
2/26/2017
3/26/2017
4/26/2017
A
D
E
F
B
C
Cross Module


Evolutionary approach with Service B
More up front design with Service E
45
40
35
30
25
20
15
10
5
0
4/26/2016
5/26/2016
6/26/2016
7/26/2016
8/26/2016
9/26/2016
10/26/2016
11/26/2016
12/26/2016
1/26/2017
2/26/2017
3/26/2017
4/26/2017
A
D
E
F
B
C
Cross Module


45
40
35
30
25
20
15
10
5
0
Introduced Cross Module ADRs
4/26/2016
5/26/2016
6/26/2016
7/26/2016
8/26/2016
9/26/2016
10/26/2016
11/26/2016
12/26/2016
1/26/2017
2/26/2017
3/26/2017
4/26/2017
A
D
E
F
B
C
Cross Module


Bursts of decisions
45
40
35
30
25
20
15
10
5
0
4/26/2016
5/26/2016
6/26/2016
7/26/2016
8/26/2016
9/26/2016
10/26/2016
11/26/2016
12/26/2016
1/26/2017
2/26/2017
3/26/2017
4/26/2017
A
D
E
F
B
C
Cross Module

Observations

- Clearly see designers who prefer up front vs. slow evolution
- Architecture as a whole emerged over time
  - Technical constraints not documented as ADRs
- Decisions come in bursts
- Architecture eventually settles
- Larger system might have more ADRs (not enough data)
- Documented modules seemed to have less rework, greater general design awareness

I wish we had greater empirical evidence that showed how ADRs effect the software system over time.

Potentially interesting focus areas

- Impact on quality
- Ability to manage technical debt
- Effects on communication
- Influence on design competence
- Quality of design decisions

How did the team like them?

- Everyone found them *useful* or *very useful*
- Most teammates referenced ADRs rarely
  - Read once or twice, months between reviews
  - Extremely helpful for onboarding new teammates
- Biggest complaints
  - Forming good documentation habits is hard
  - Architecture design vs. detailed design
  - Discovering ADRs when you don’t know to look for them
- Biggest likes
  - History of the project
  - Great practice for improving technical thinking

ADVICE FOR EFFECTIVE ADRS

Store ADRs with the code in plain text

- Easy to consume, easy to change
- Integrates with peer review workflow and tools
- Contextually close to where it’s used

- Problem: Where do cross cutting concerns go?
  - Our solution: create a dedicated “Architecture” repository

Delegate ADR Creation

- Grow the team’s design skills
- Small responsibility, little risk
- Easy to review
- Opportunity for training
  - Peer review, pair, coach

Peer Review as you would Code

- Get the whole team involved
- Spread knowledge
- Allow team to make decisions without architect

Foster a documentation habit

- Architect points out when a decision is made
  - “Do we need to record an ADR?”
- Track ADRs in the backlog
  - Hold the team accountable
- Make templates readily available
- Use architecture briefings

Make a decision, then document it

- Proposed decisions without consensus thrashed during peer review
- Document and share a decision
- Remember ideas for the future

Not everything is an ADR!

- Team loved ADRs
  - At first, used them for anything and everything
- Lightweight, text-based, single responsibility “records”
  - Views
  - Design guidance
  - Governance
  - Stakeholder, quality attribute viewpoints

ADR Tips and Advice

- Store with the code. Use plain text.
- Delegate ADR creation.
- Peer review as you would code.
- Foster a documentation habit.
- Make a decision, then document it.
- Not everything is an ADR!

DISCUSSION

ADRs are awesome but not new.

Architecture Decisions… cool since at least 1997

- **1997**: Architecture in Practice, first edition
  - Len Bass, Paul Clements, Rick Kazman
- **2002**: Documenting Software Architecture: Views and Beyond, first edition
  - Paul Clements et al.
- **2005**: Architecture Decisions: Demystifying Architecture
  - Jeff Tyree and Art Akerman
- **2009**: The Decision View's Role in Software Architecture Practice
  - Phillipe Krutchten
- **2011**: A documentation framework for architecture decisions
  - Uwe Van Heesch, Paris Avgerioum, and Rich Hilliard
- **2011**: **Documenting Architecture Decisions**,
  - http://thinkrelevance.com/blog/2011/11/15/documenting-architecture-decisions
  - Michael Nygard

Why is this popular now?

- Cultural shift among new developers
  - Expect to be included in the design process
- Increased system complexity leads to greater modularity
  - Need for better architectural thinking at scale
  - “I can’t do this alone”
- Democratization of design authority
- Shift toward architect as coach or mentor

Documenting design decisions is a no brainer.

“Please document your design decisions…”

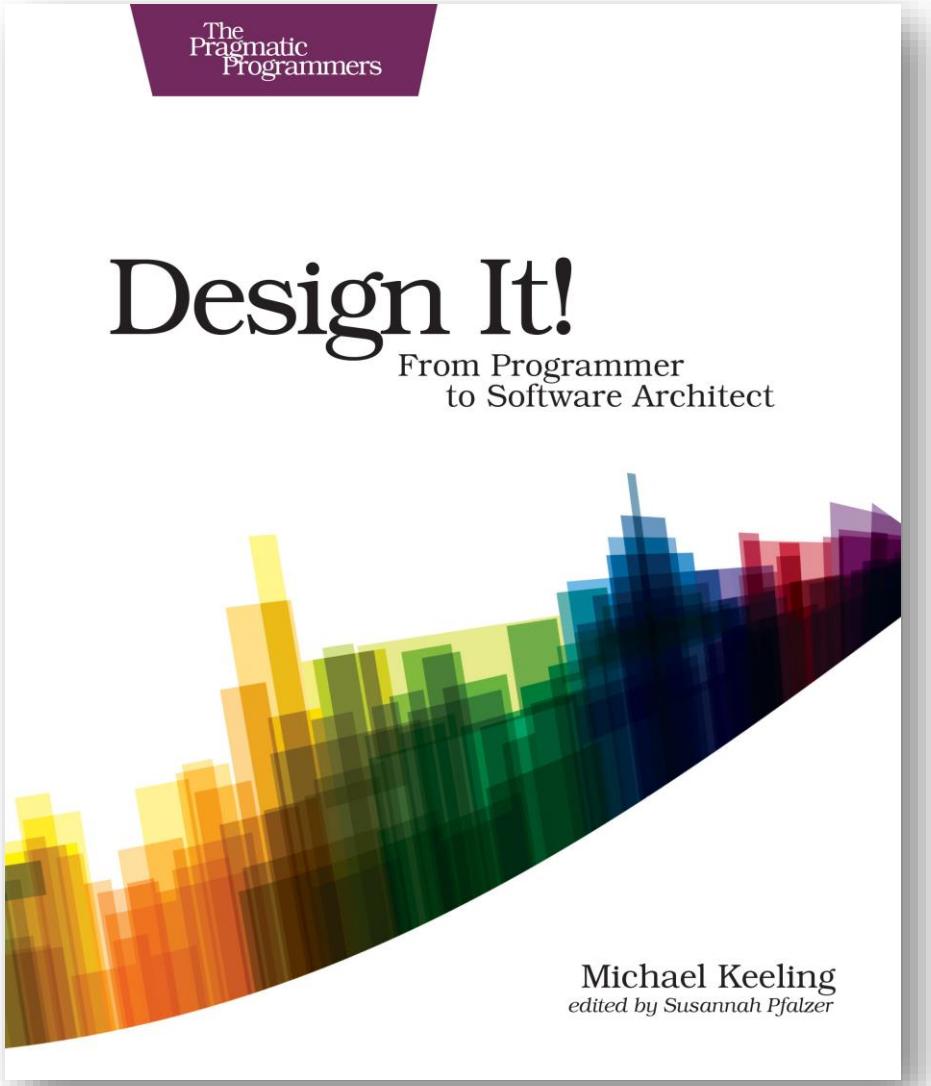
Get started with ADRs on your team!

You don’t need permission to start!

Step 1: Create a template
Step 2: Create your team’s first ADR
Step 3: Ask someone to review it

Silver Toolbox

Thank you!



Michael Keeling
@michaelkeeling
neverletdown.net

Joe Runde
@joerunde

Buy Design It! now at
http://bit.ly/2pJOrly