The future is now

the AI assisted programming paradigm shift

Software Engineer @GitHub

I work on **GitHub Actions**

I **create technical content** in my free time by working 7 days a week

Born and raised in a **rural Lebanese town**

I emigrated to the **Netherlands more than 5 years ago**

Yes, **I speak Arabic**

My partner is a Dutch psychologist, and no she does not psychoanalyse me (as far as I know)

Do we *discover* the future?

Do we *create* the future?

Do we *recreate* the past?

I’m going to make an argument based on 3 premises.

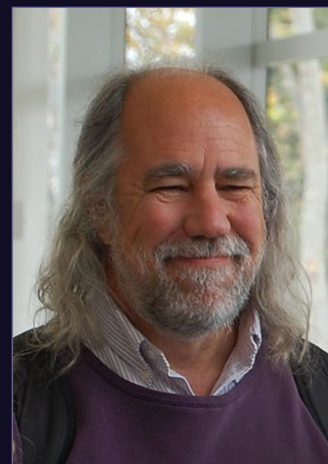
The first premise comes from a journey through the past.

Grady Booch

IBM Chief Scientist of Software Engineer

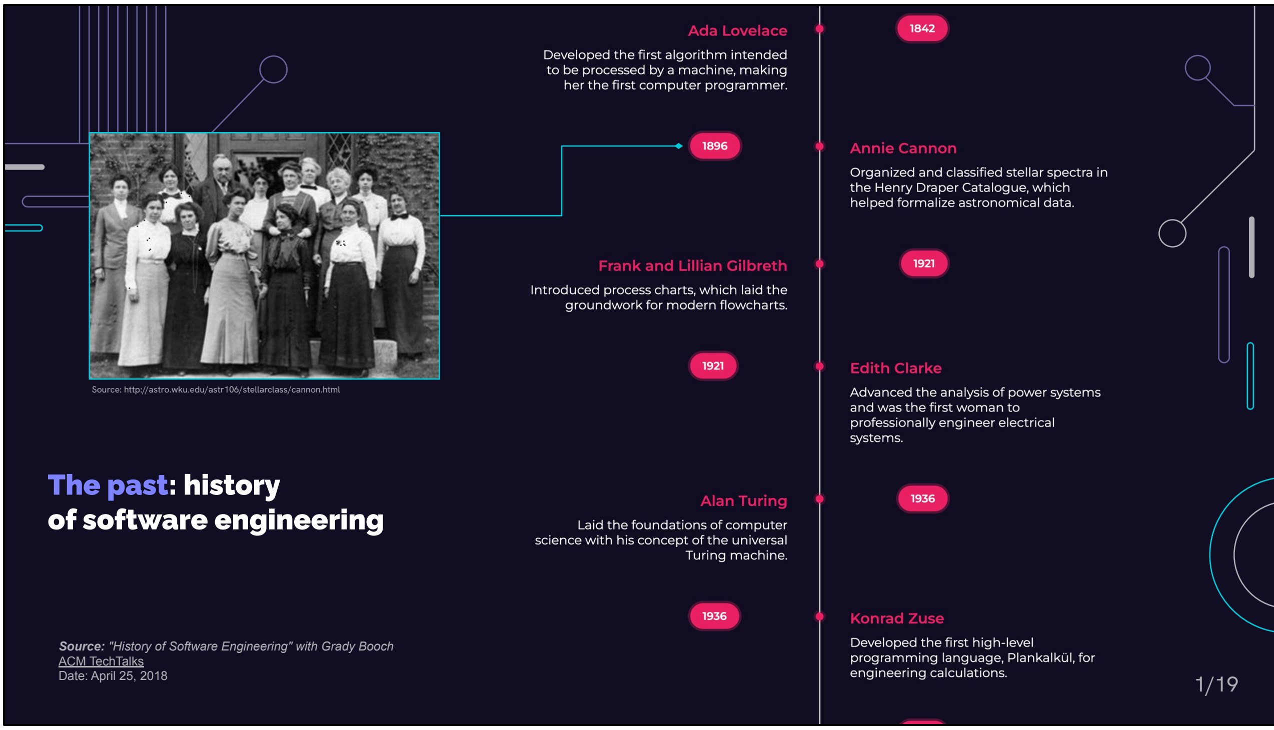
What follows is a rendition of his (and team) work on the story of computing

https://computingthehumanexperience.com/



No aspect of this talk would have been possible without the seminar work of Grady Booch. All credit for cataloguing the history of computing goes to him.

The following is a rendition of Grady’s work that is cherry picking some events through time.



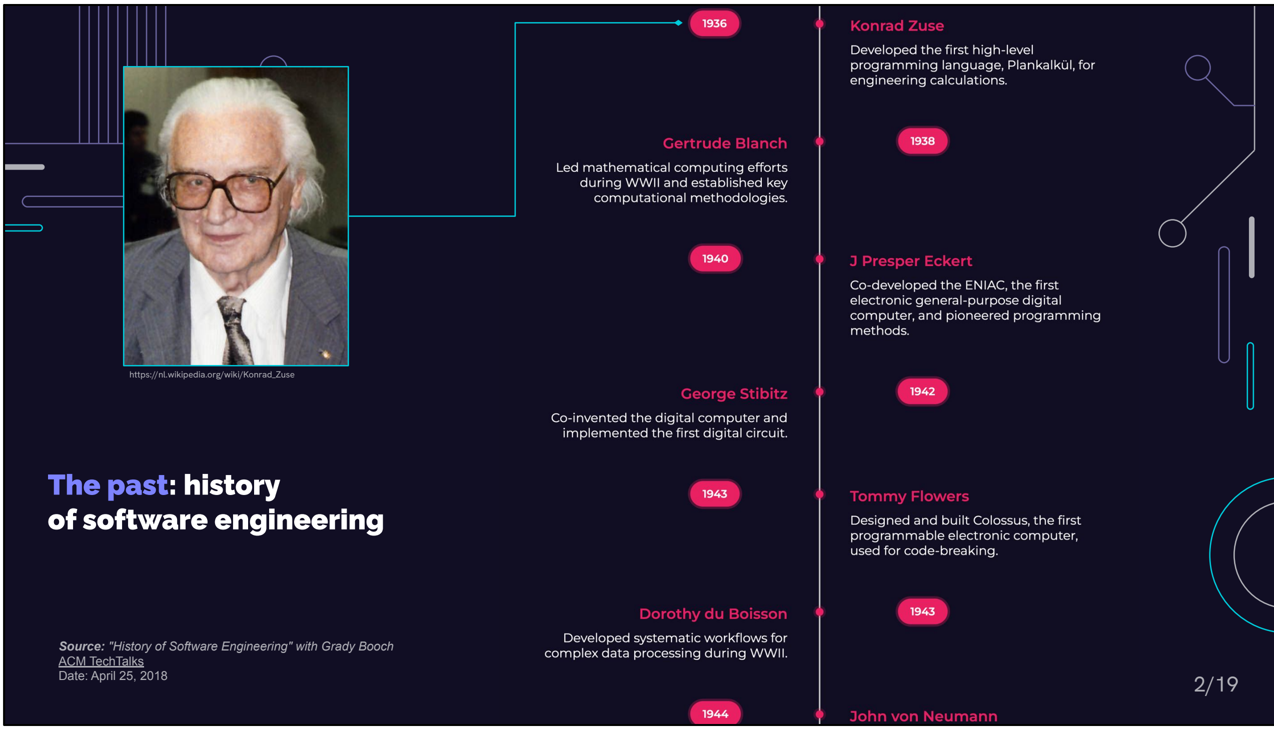
Annie Cannon and the Harvard Computers.

Yes, in 1896, computers was the term to refer to team of women working at the Harvard College Observatory, to process astronomical data. They were hired by Charles Pickering, who quite misogynistically thought that the work these ladies did was beneath him.

Cannon first started cataloging the stars, she was able to classify **1,000 stars in three years**, but by 1913, she was able to work on **200 stars an hour**.

Cannon could classify **three stars a minute** just by looking at their spectral patterns and, if using a magnifying glass, could classify stars down to the ninth magnitude, around 16 times fainter than the human eye can see.

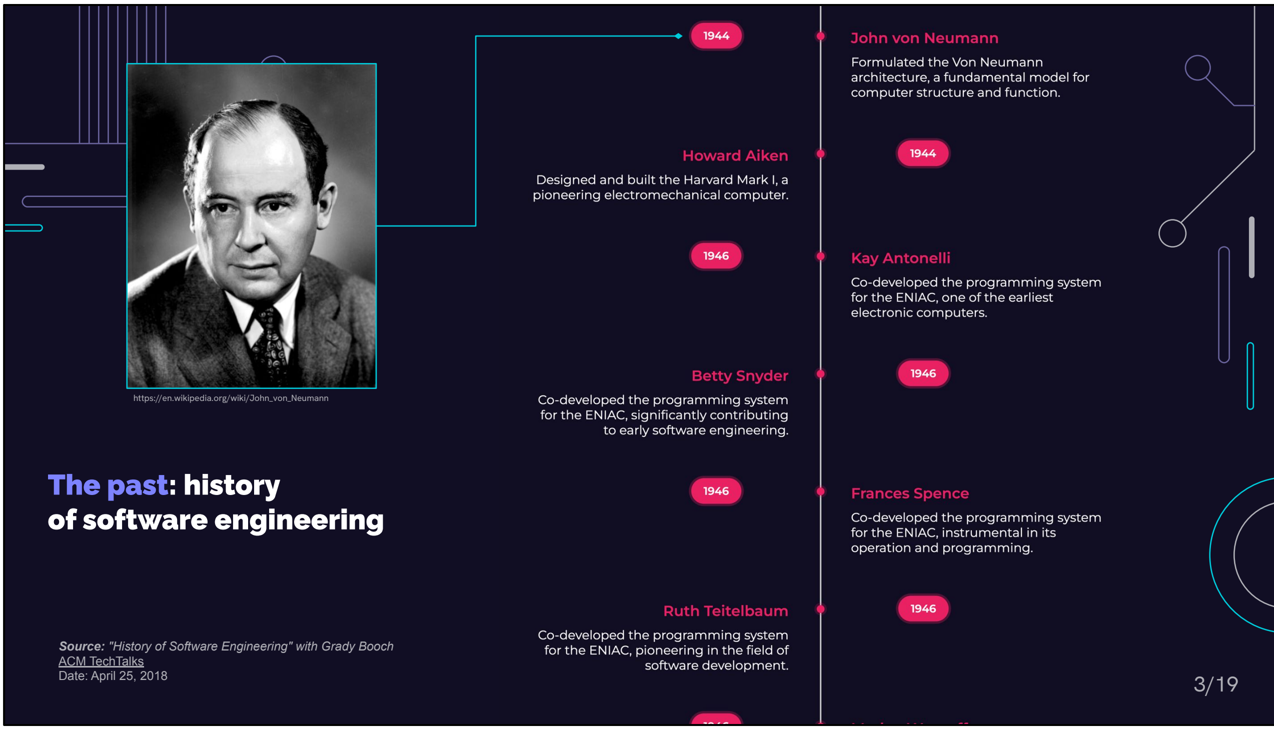
Her work was also highly accurate.



Zuse created the world's first programmable computer; the functional program-controlled Turing-complete Z3 became operational in May 1941.

In 1941, he founded one of the earliest computer businesses, producing the Z4, which became the world's first commercial computer.

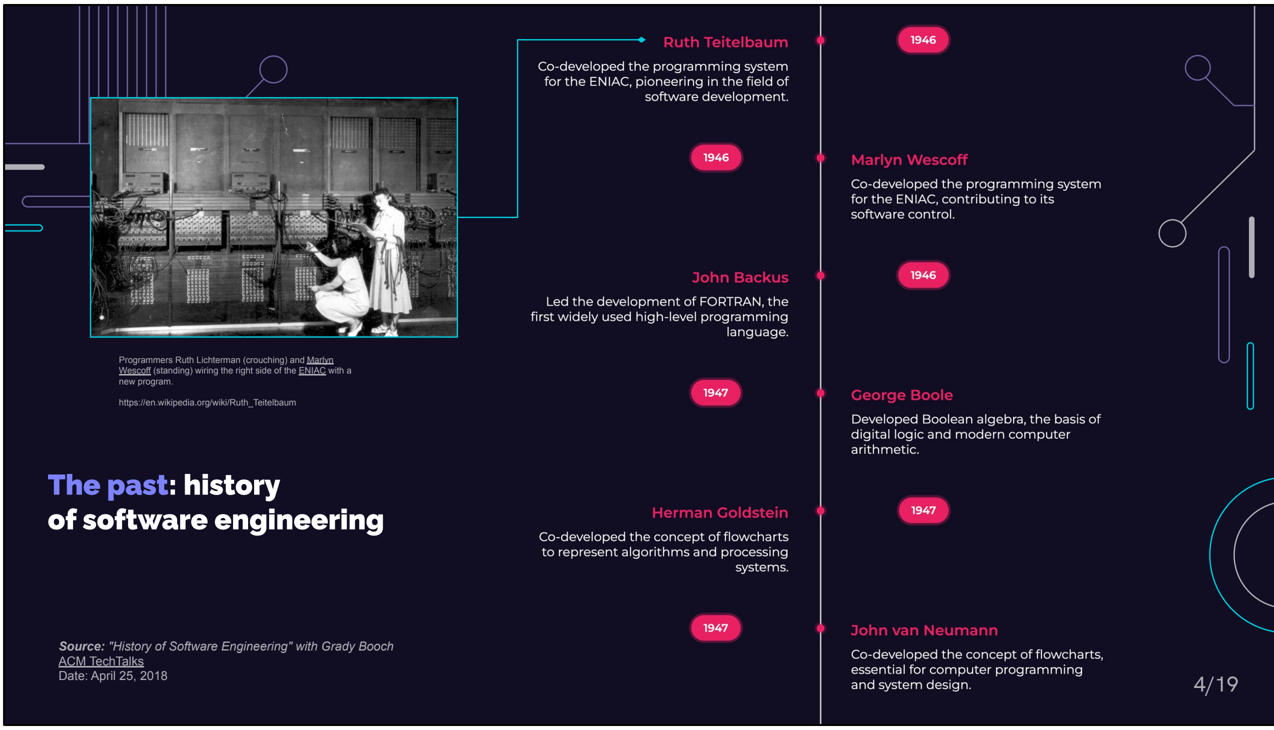
He also created Plankalkül the first high level programming language designed for a computer.



John von Neumann was a polymath by every definition of the term.

His contributions to computer science, mathematics, physics and engineering are countless.

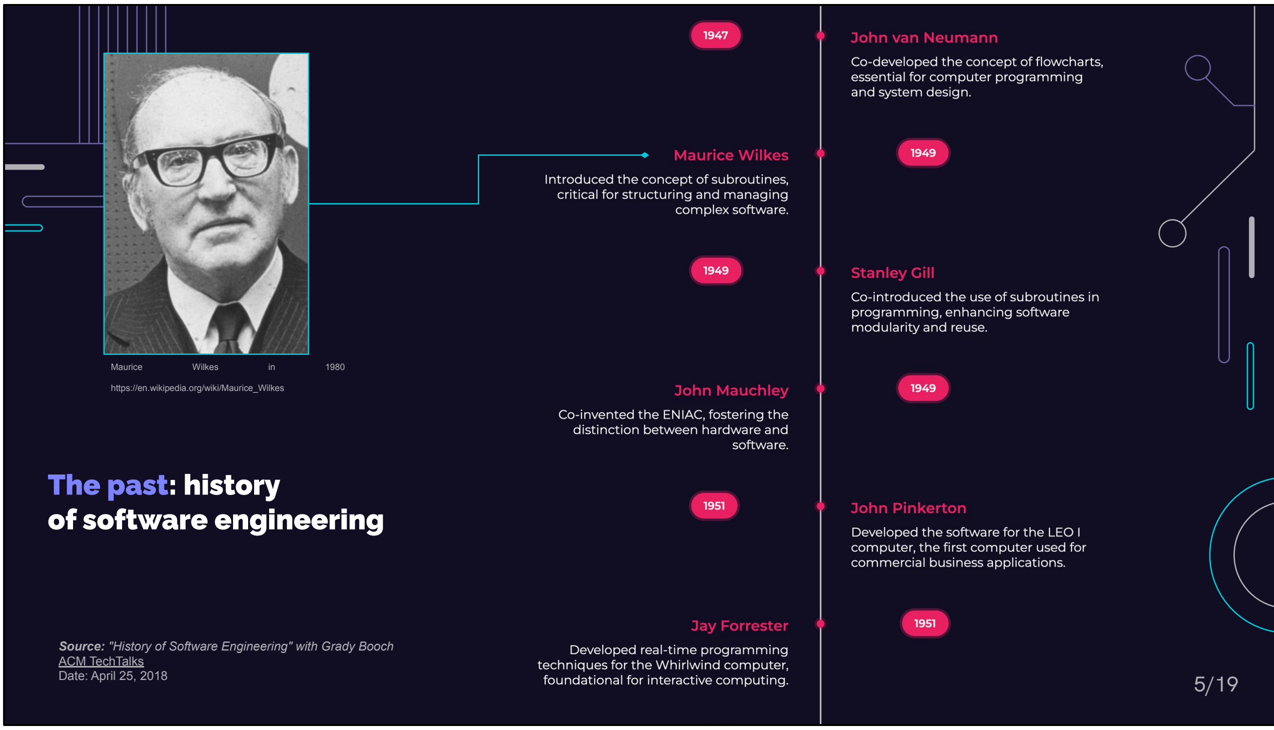
He’s listed here specifically for the creation and evolution of the von Neumann architecture which became the basis of all modern computers.



ENIAC: Electronic Numerical Integrator and Computer was the result of a U.S. government-funded project during World War II to build an electronic computer that could be programmed.

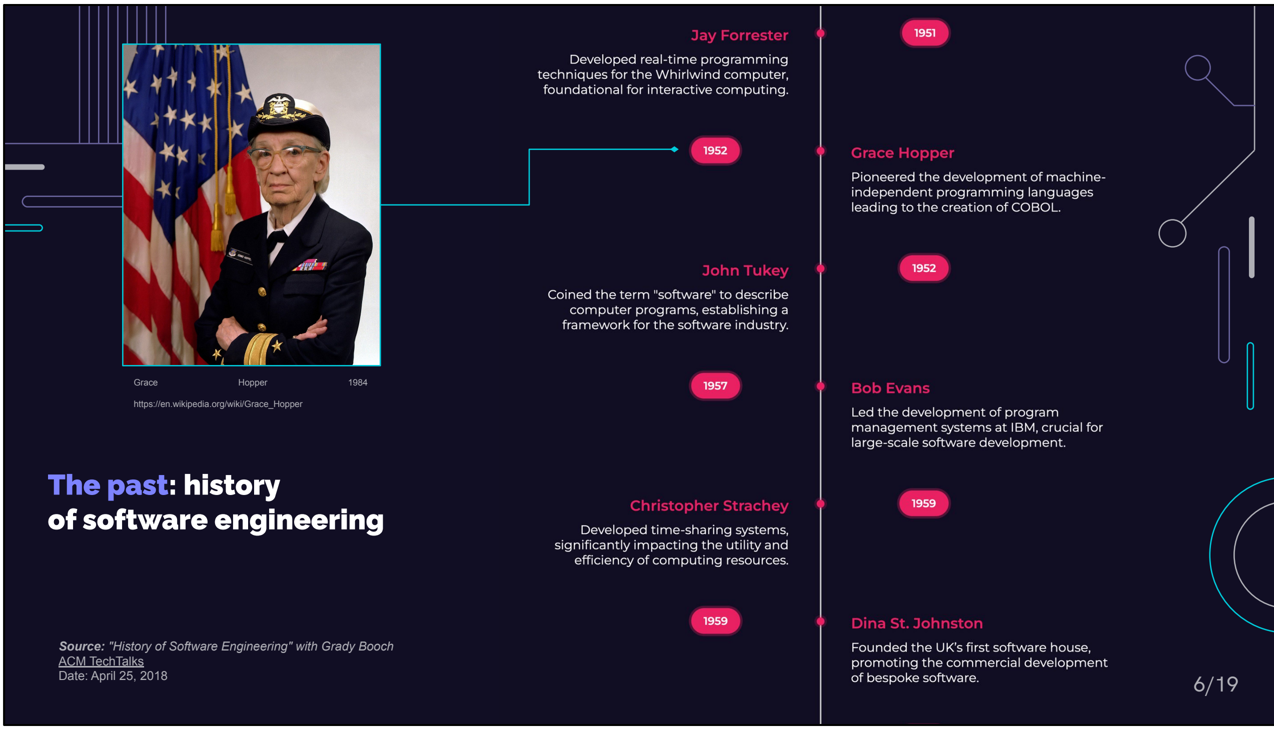
Ruth Teitelbaum was one of the original programmers for the ENIAC computer.

You can see her crouching in this picture.

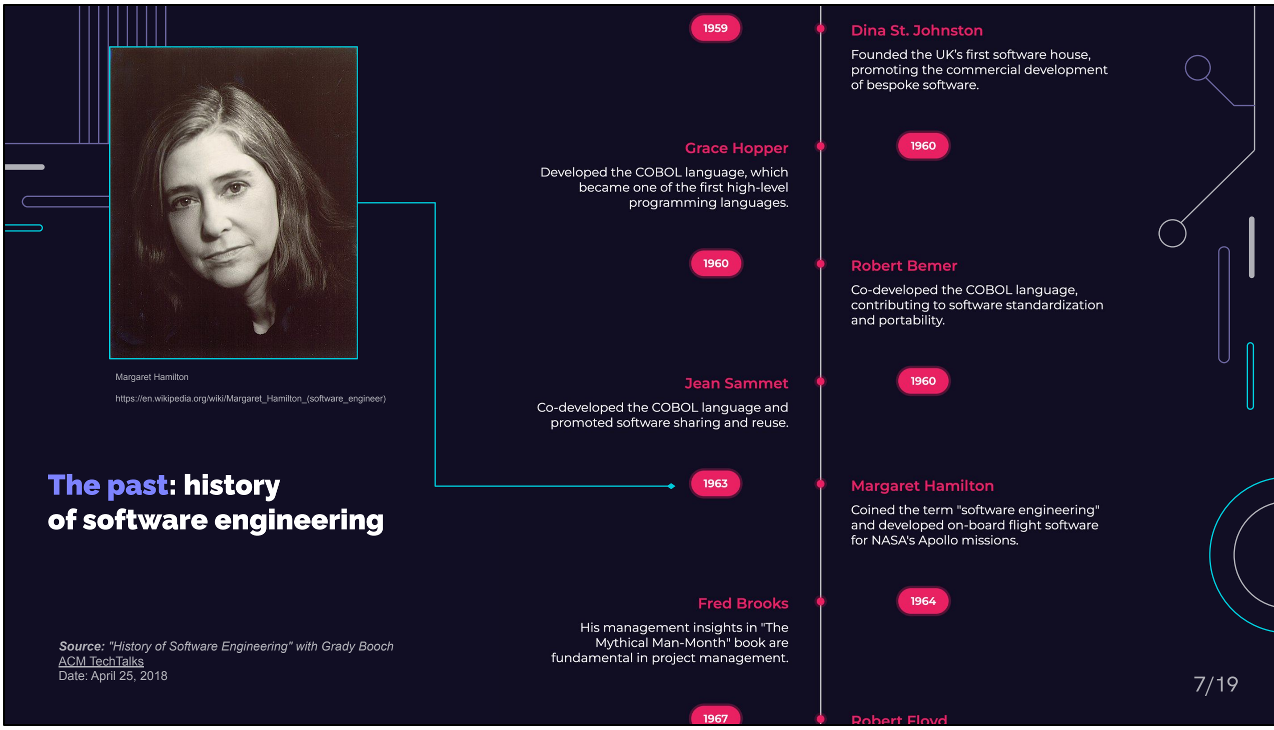


Designed and helped build the [Electronic Delay Storage Automatic Calculator]

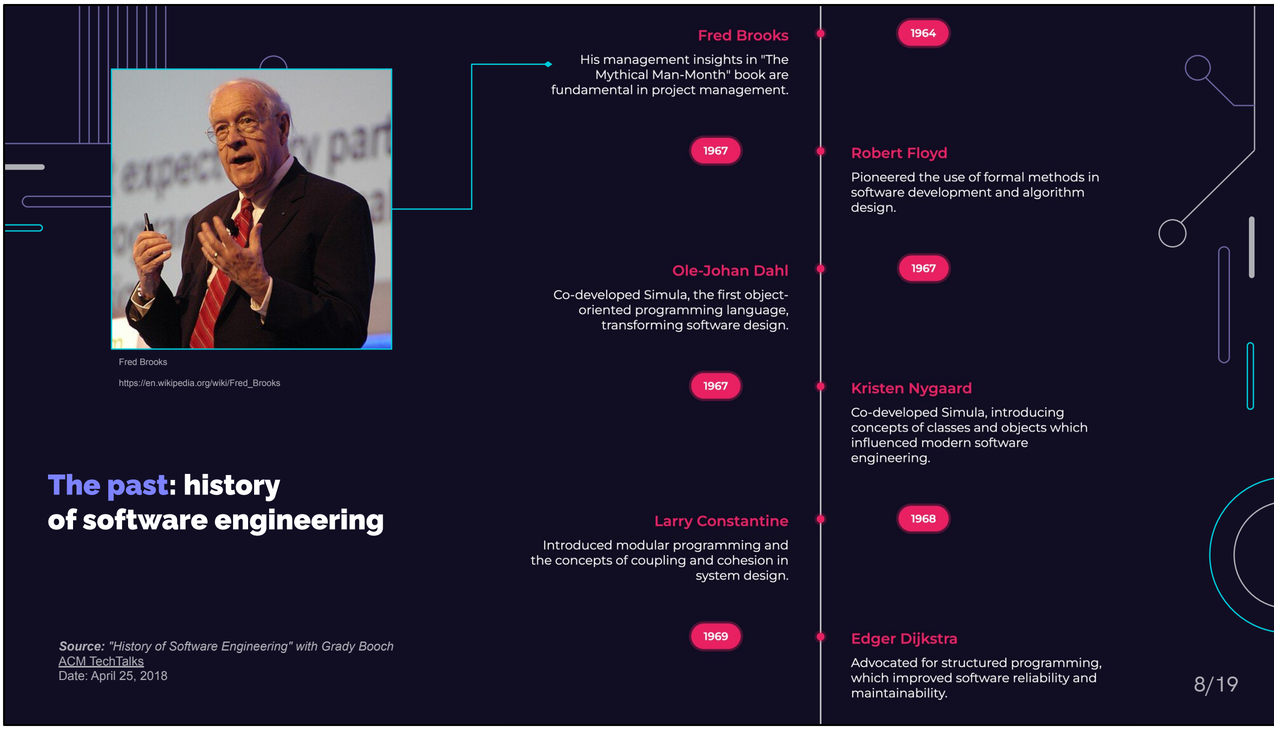
symbolic labels, [macros] and subroutine libraries



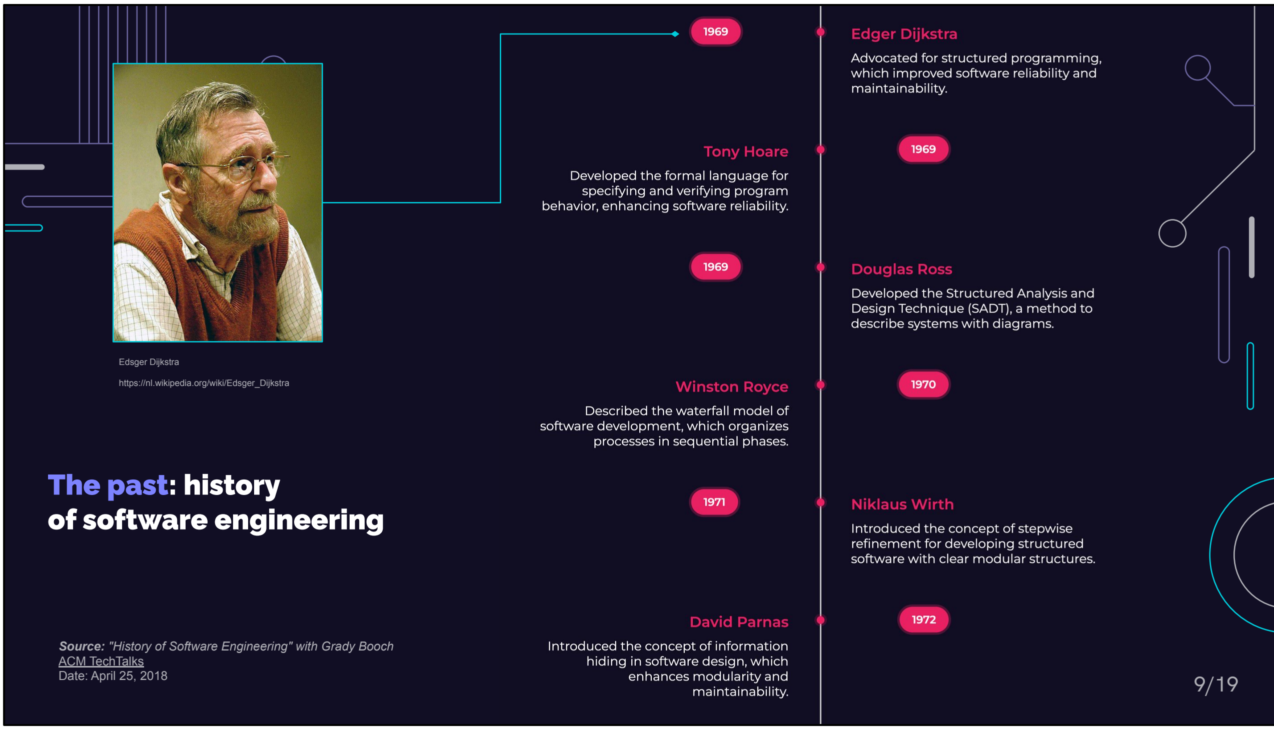
The first to devise the theory of machine-independent programming languages



Hamilton introduced the term Software Engineering to our vocabulary

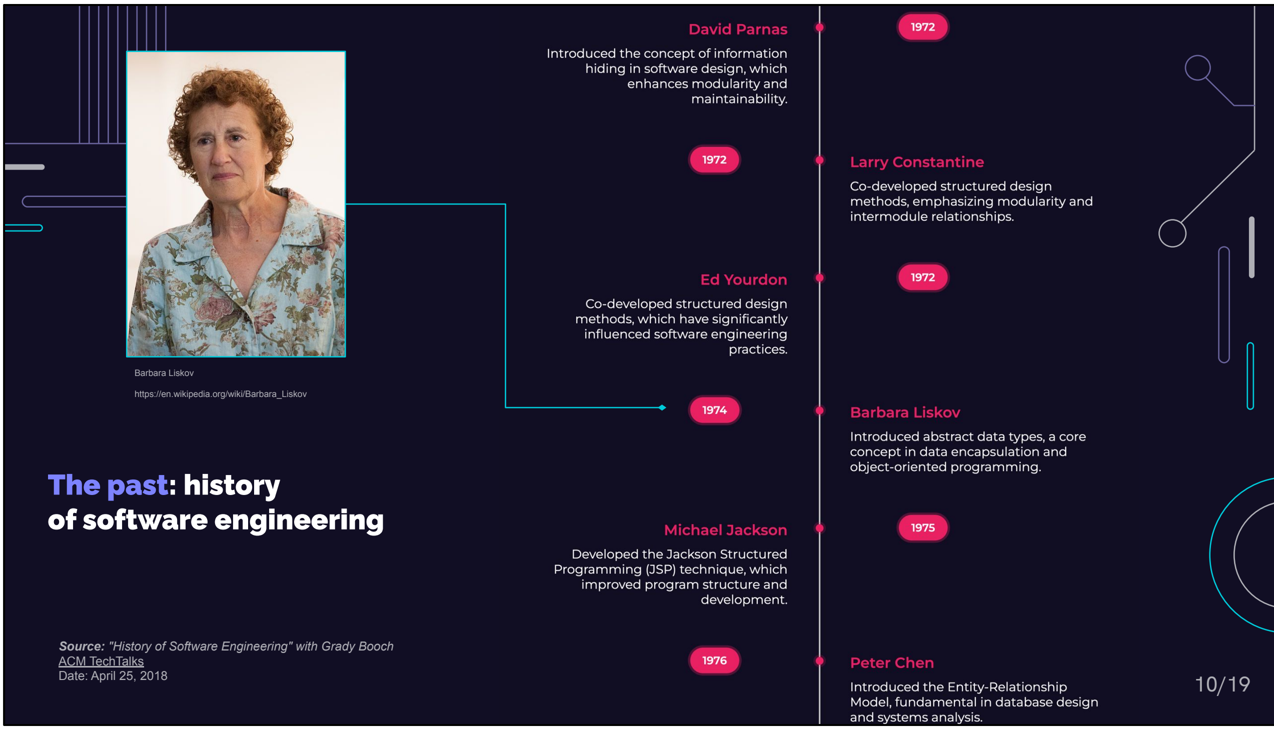


The mythical man-month, the book that everyone tells you to read



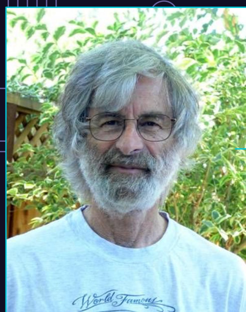
Introduced structured programming:

- "Sequence"; ordered statements or subroutines executed in sequence.
- "Selection"; one or a number of statements is executed depending on the state of the program.
    a. This is usually expressed with keywords such as if..then..else..endif. The conditional statement should have at least one true condition and each condition should have one exit point at max.
- "Iteration"; a statement or block is executed until the program reaches a certain state



Introduced abstract data types: Collection Container List String Set Multiset Map Multimap Graph Tree Stack Queue Priority queue Double-ended queue Double-ended priority queue

SOLID: Liskov substitution principle: https://en.wikipedia.org/wiki/Liskov_substitution_principle



Leslie Lamport

https://en.wikipedia.org/wiki/Leslie_Lamport

The past: history of software engineering

1976 — Peter Chen
Introduced the Entity-Relationship Model, fundamental in database design and systems analysis.

1976 — Michael Fagan
Introduced software inspections, which significantly improve software quality and reduce defects.

1977 — John Backus
Developed functional programming concepts with the introduction of the programming language FP.

1978 — Tom DeMarco
Advocated structured analysis and system specification in software engineering, influencing project management.

1978 — Leslie Lamport
Developed LaTeX and contributed to the theory and practice of distributed systems.

1983 — Richard Stallman
Founded the Free Software Movement, advocating for the freedom to use, study, distribute, and modify software.

Source: *"History of Software Engineering" with Grady Booch*
ACM TechTalks
Date: April 25, 2018

The past: history of software engineering

Brad Cox

https://www.furman.edu/news/computer-pioneer-brad-cox-65-passes-away/

1983 — Richard Stallman

Founded the Free Software Movement, advocating for the freedom to use, study, distribute, and modify software.

1986 — Grady Booch

Developed the Booch method for software engineering, which later evolved into the Unified Modeling Language (UML).

1986 — Victor Basili

Pioneered empirical software engineering, applying scientific methods to software development and assessment.

1986 — Brad Cox

Developed Objective-C, influencing component-based software engineering and the development of reusable components.

1987 — Harlan Mills

Introduced the cleanroom software engineering method, aiming for zero-defect software through formal methods.

1988 — Stephen Mellor

Promoted object-oriented analysis, influencing the development of analysis

Source: *"History of Software Engineering" with Grady Booch*
ACM TechTalks
Date: April 25, 2018

Ed Yourdon

https://en.wikipedia.org/wiki/Edward_Yourdon

The past: history of software engineering

Source: *"History of Software Engineering" with Grady Booch*
ACM TechTalks
Date: April 25, 2018

1987 – Harlan Mills
Introduced the cleanroom software engineering method, aiming for zero-defect software through formal methods.

1988 – Stephen Mellor
Promoted object-oriented analysis, influencing the development of analysis and design methodologies.

1988 – Barry Boehm
Developed the spiral model of software development, integrating iterative development with systematic controls.

1988 – Watts Humphrey
Developed the Capability Maturity Model, which provides a framework for assessing and improving software processes.

1989 – Ed Yourdon
Developed the structured analysis technique, a methodological approach to object-oriented programming and design.

1989 – Rebecca Wirfs-Brock
Pioneered responsibility-driven design, enhancing the design of object-oriented software.

The past: history of software engineering

Erich Gamma

1989
Rebecca Wirfs-Brock
Pioneered responsibility-driven design, enhancing the design of object-oriented software.

1990
Jim Rumbaugh
Developed the Object Modeling Technique (OMT), which is an object-oriented analysis and design technique.

1990
Ivar Jacobson
Developed the Objectory methodology for software development, emphasizing use case driven development.

1990
Peter Coad
Promoted object-oriented analysis and design, enhancing software architectural practices.

1991
Alan Cooper
Pioneered visual programming with Visual Basic, drastically simplifying the programming experience.

1994
Erich Gamma
Co-authored "Design Patterns: Elements of Reusable Object-Oriented Software," which has influenced software design.

Source: *"History of Software Engineering" with Grady Booch*
ACM TechTalks
Date: April 25, 2018

Mary Shaw

https://en.wikipedia.org/wiki/Mary_Shaw_(computer_scientist)

The past: history of software engineering

1991 — Alan Cooper

Pioneered visual programming with Visual Basic, drastically simplifying the programming experience.

1994 — Erich Gamma

Co-authored "Design Patterns: Elements of Reusable Object-Oriented Software," which has influenced software design.

1995 — Jeff Sutherland

Co-created Scrum, a framework for agile software development, fostering iterative and incremental processes.

1995 — Philippe Kruchten

Developed the Rational Unified Process, a comprehensive software development process framework.

1995 — Mary Shaw

Advocated for recognizing software architecture as a distinct discipline within software engineering.

1996 — Kent Beck

Developed Extreme Programming (XP), a methodology that emphasizes customer satisfaction and rapid development.

Source: *"History of Software Engineering" with Grady Booch*
ACM TechTalks
Date: April 25, 2018

Jeff Dean

https://twitter.com/JeffDean

The past: history of software engineering

Kent Beck — 1996

Developed Extreme Programming (XP), a methodology that emphasizes customer satisfaction and rapid development.

Eric Raymond — 1997

Advocated for open source software, highlighting the collaborative nature of software development.

Martin Fowler — 1999

Introduced the concept of refactoring in software engineering, improving the design of existing code.

Walker Royce — 2000

Enhanced the Rational Unified Process with iterative development principles, improving software project success.

Jeff Dean — 2000

Pioneered innovations in scalable systems at Google, significantly impacting cloud computing and big data.

Roy Fielding — 2000

Described the Representational State Transfer (REST) architectural style, foundational to modern web development.

Source: *"History of Software Engineering" with Grady Booch*
ACM TechTalks
Date: April 25, 2018

Kent Beck

https://en.wikipedia.org/wiki/Kent_Beck

The past: history of software engineering

Source: *"History of Software Engineering" with Grady Booch*
ACM TechTalks
Date: April 25, 2018

2000

Roy Fielding

Described the Representational State Transfer (REST) architectural style, foundational to modern web development.

2003

Kent Beck

Developed Test-Driven Development (TDD), a methodology that integrates testing and development to improve software quality and responsiveness to change.

2003

Eric Evans

Introduced Domain-Driven Design (DDD), a framework for developing complex software systems that meet specific business requirements.

2003

Gregor Hohpe

Co-authored "Enterprise Integration Patterns", providing a standard language and set of best practices for integration strategies in complex systems.

2003

Bobby Woolf

Co-authored "Enterprise Integration Patterns", helping professionals navigate and implement effective integration solutions in software architecture.

2003

Mary Poppendieck

Linus Torvalds

https://en.wikipedia.org/wiki/Linus_Torvalds

The past: history of software engineering

Source: *"History of Software Engineering" with Grady Booch*
ACM TechTalks
Date: April 25, 2018

2003 – Mary Poppendieck
Co-developed Lean Software Development, adapting lean manufacturing principles to software development to optimize efficiency and quality.

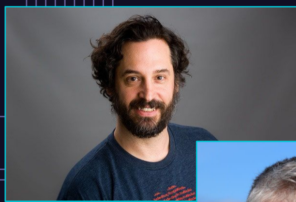
2003 – Tom Poppendieck
Co-developed Lean Software Development, providing a methodology for streamlining production processes in software engineering.

2005 – Linus Torvalds
Created Git, a distributed version control system essential for software development collaboration.

2005 – Jim Coplien
Developed organizational patterns for software development, enhancing team structures and interactions.

2006 – Jeannette Wing
Promoted computational thinking, integrating problem-solving skills across disciplines.

2006 – Jeff Bezos
Developed Amazon Web Services, pioneering cloud computing platforms



Andrew Shafer

Patrick Dubois

The past: history of software engineering

Jeannette Wing — 2006

Promoted computational thinking, integrating problem-solving skills across disciplines.

Jeff Bezos — 2006

Developed Amazon Web Services, pioneering cloud computing platforms that revolutionized how applications are deployed and managed.

Joel Spolsky — 2007

Co-created Stack Overflow, a platform for software developers to share knowledge and improve coding skills.

Robert Martin — 2008

Advocated for clean code principles, promoting better coding practices for maintainable and efficient software.

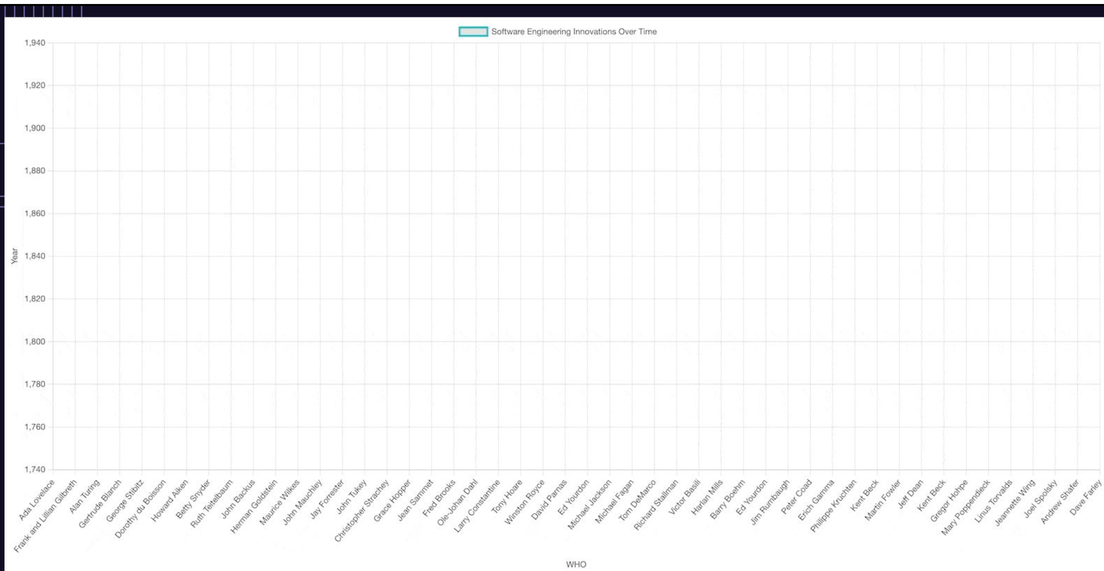
Andrew Shafer — 2008

Co-introduced the concept of DevOps, integrating software development and operations for faster delivery.

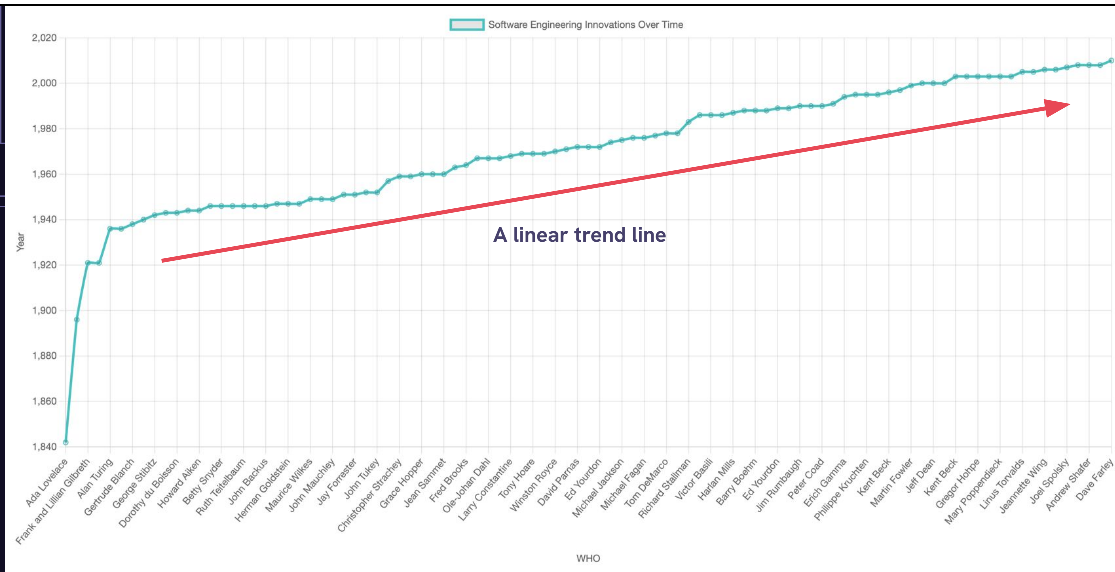
Patrick Debois — 2008

Co-introduced the concept of DevOps, emphasizing collaboration and automation in software development.

Source: *"History of Software Engineering" with Grady Booch*
ACM TechTalks
Date: April 25, 2018



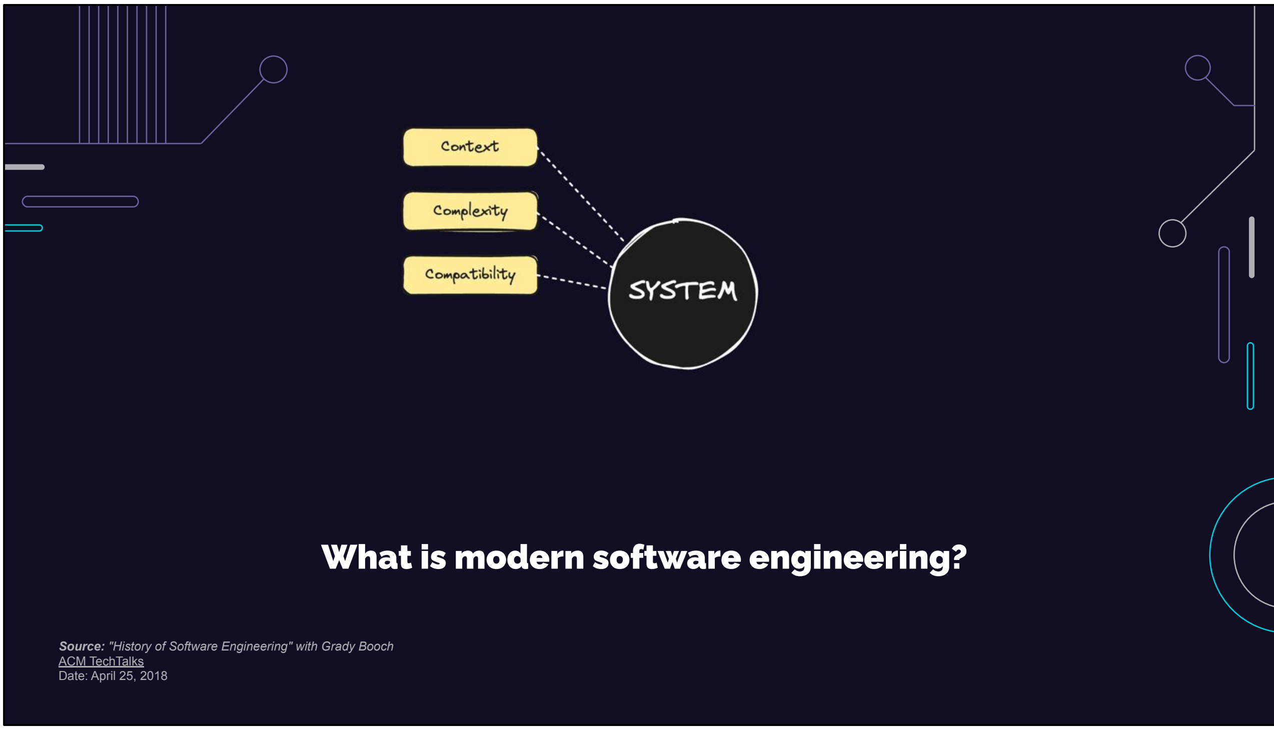
The past: history of software engineering



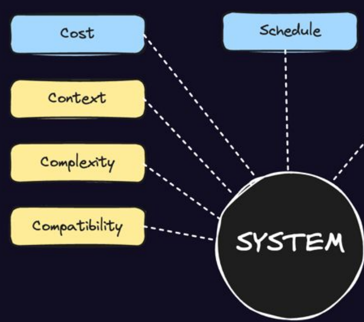
The past: history of software engineering

This is the first premise. Keep it in mind as we discuss the present.

What is Modern Software Engineering?

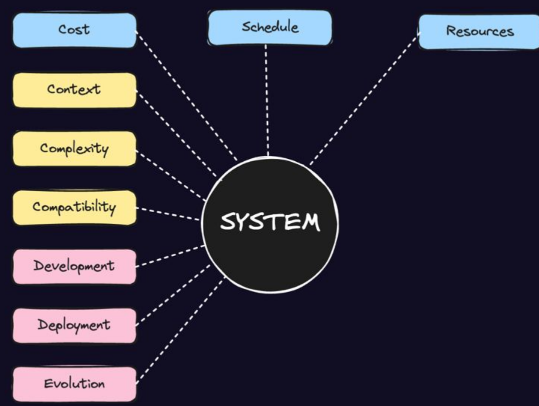


The system in the middle undergoes changes as a direct result of different forces applied to it.



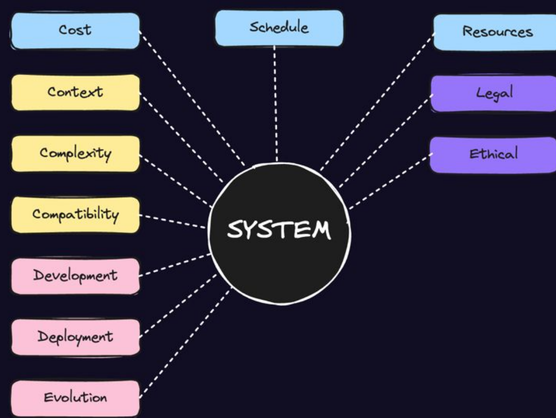
What is modern software engineering?

Source: *"History of Software Engineering" with Grady Booch*
ACM TechTalks
Date: April 25, 2018



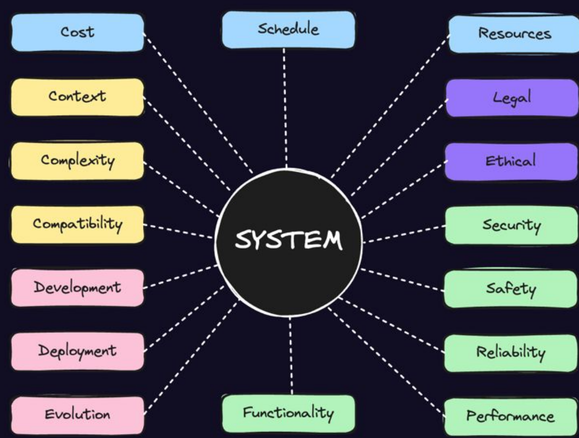
What is modern software engineering?

Source: *"History of Software Engineering" with Grady Booch*
ACM TechTalks
Date: April 25, 2018



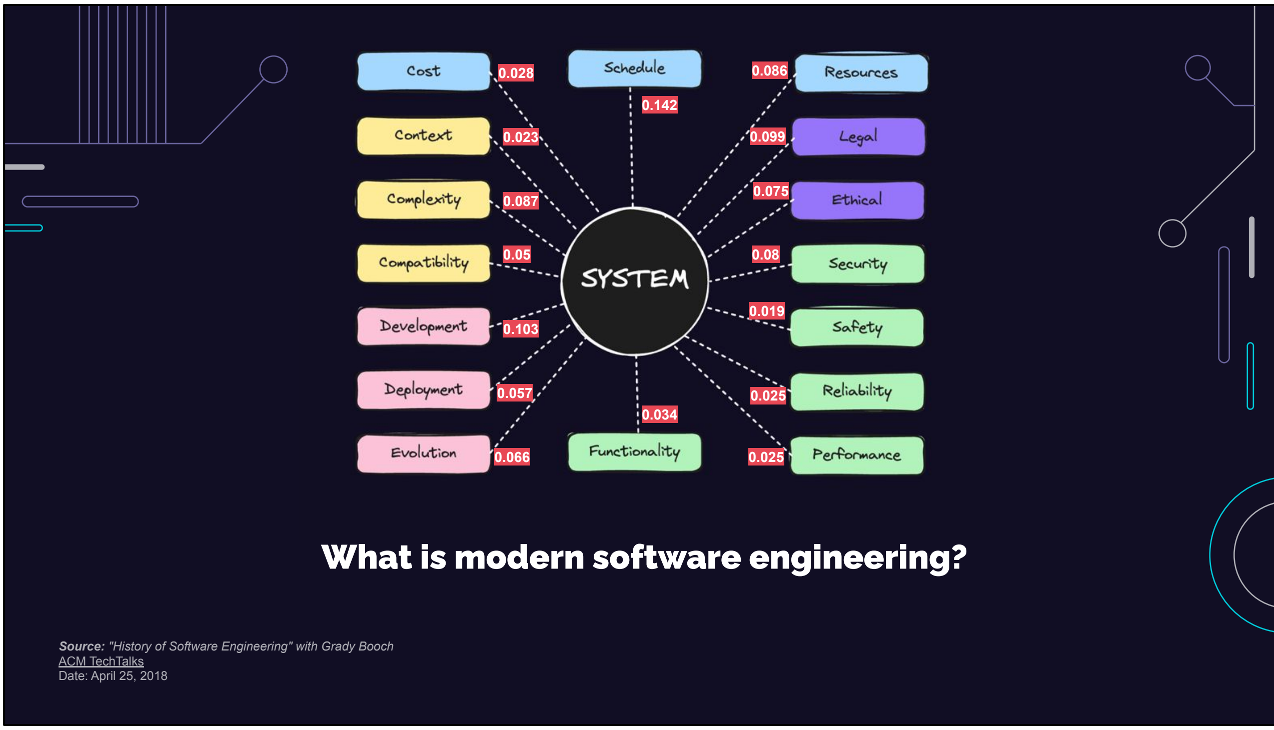
What is modern software engineering?

Source: *"History of Software Engineering" with Grady Booch*
ACM TechTalks
Date: April 25, 2018



What is modern software engineering?

Source: *"History of Software Engineering" with Grady Booch*
ACM TechTalks
Date: April 25, 2018



The numbers in red (totally fictional and have no basis in reality) represent the weights of each of the forces applied on a given system.

These weights vary by industry vertical, geography, company type, funding sources, business maturity levels etc.

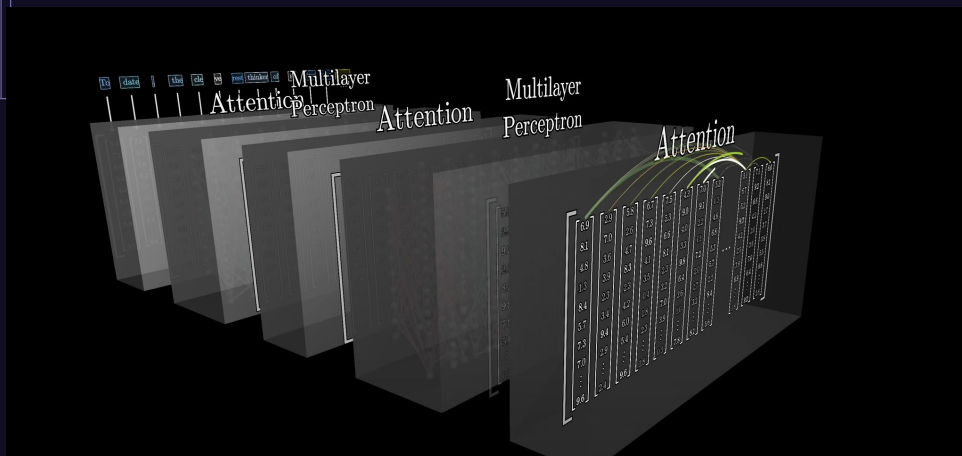
What matters is that all these forces are interdependent. A change to one will definitely affect the other.

The dependence function is in constant flux. A single combination of these weights at a given point in time constitutes the fingerprint of a system.

Our job as software engineers is to maintain our systems in a state of homeostasis, a state of equilibrium as the weights of these forces are constantly changing.

It’s not about the tech, it never was.

This is the second premise.



How much of software engineering can LLMs do?

Source: "Attention in transformers, visually explained | Chapter 6, Deep Learning"
3Blue1Brown
Date: April 7, 2024

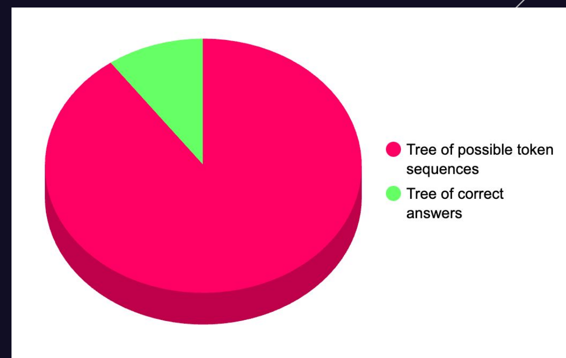
Auto-regressive LLMs have problems:

1. Auto-regressive LLMs **cannot be made factual**, and toxicity can be filtered but not entirely eliminated

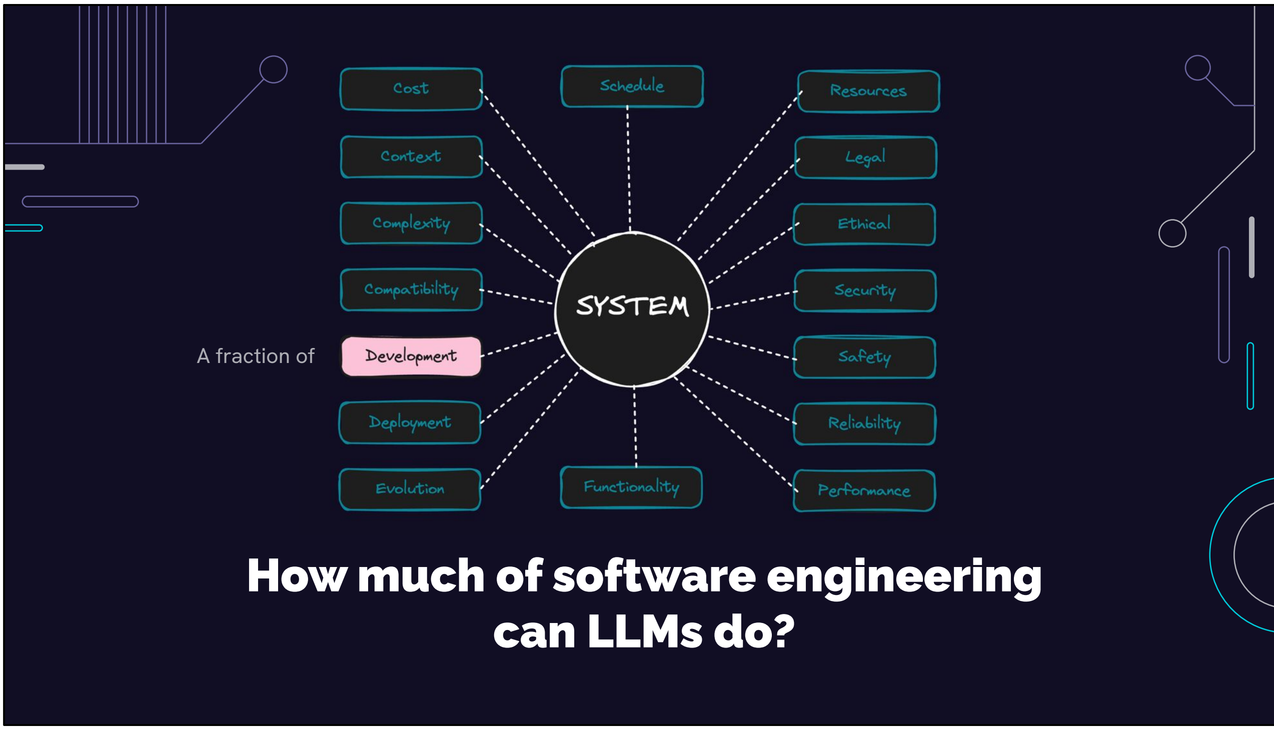
2. They are not controllable

3. The **probability** that produced tokens can **diverge us to outside the set of correct answers** is high
   a. P(correct) = (1-e)^n

4. They use a **fixed amount of computation** per token

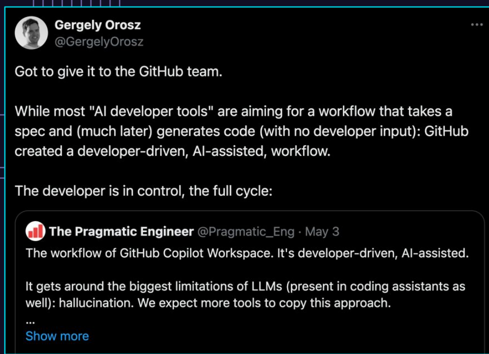


Source: "Towards AI systems that can learn, remember, reason, plan, have common sense, yet are steerable and safe"
Yann LeCun
Date: March 28, 2024


Cost
Schedule
Resources
Context
Legal
Complexity
Ethical
Compatibility
SYSTEM
Security
A fraction of
Development
Safety
Deployment
Reliability
Evolution
Functionality
Performance
How much of software engineering
can LLMs do?

Auto-regressive LLMs are still very useful!

1. They’ve proven to be **great programming companions**
2. They can **remix different forms of art**
3. They can help you with your **mental blocks**
4. They’re very helpful to **non-native speakers**



Replacing is not the objective

Gergely Orosz
@GergelyOrosz

Got to give it to the GitHub team.

While most "AI developer tools" are aiming for a workflow that takes a spec and (much later) generates code (with no developer input): GitHub created a developer-driven, AI-assisted, workflow.

The developer is in control, the full cycle:

> **The Pragmatic Engineer** @Pragmatic_Eng · May 3
> The workflow of GitHub Copilot Workspace. It's developer-driven, AI-assisted.
>
> It gets around the biggest limitations of LLMs (present in coding assistants as well): hallucination. We expect more tools to copy this approach.
> ...
> Show more

Kyle Daigle · Following
COO @ GitHub | Operationalizing AI for every employee & enabling ...
Visit my website
18h ·

The secret to AI isn't creativity, it's reducing toil. Why?

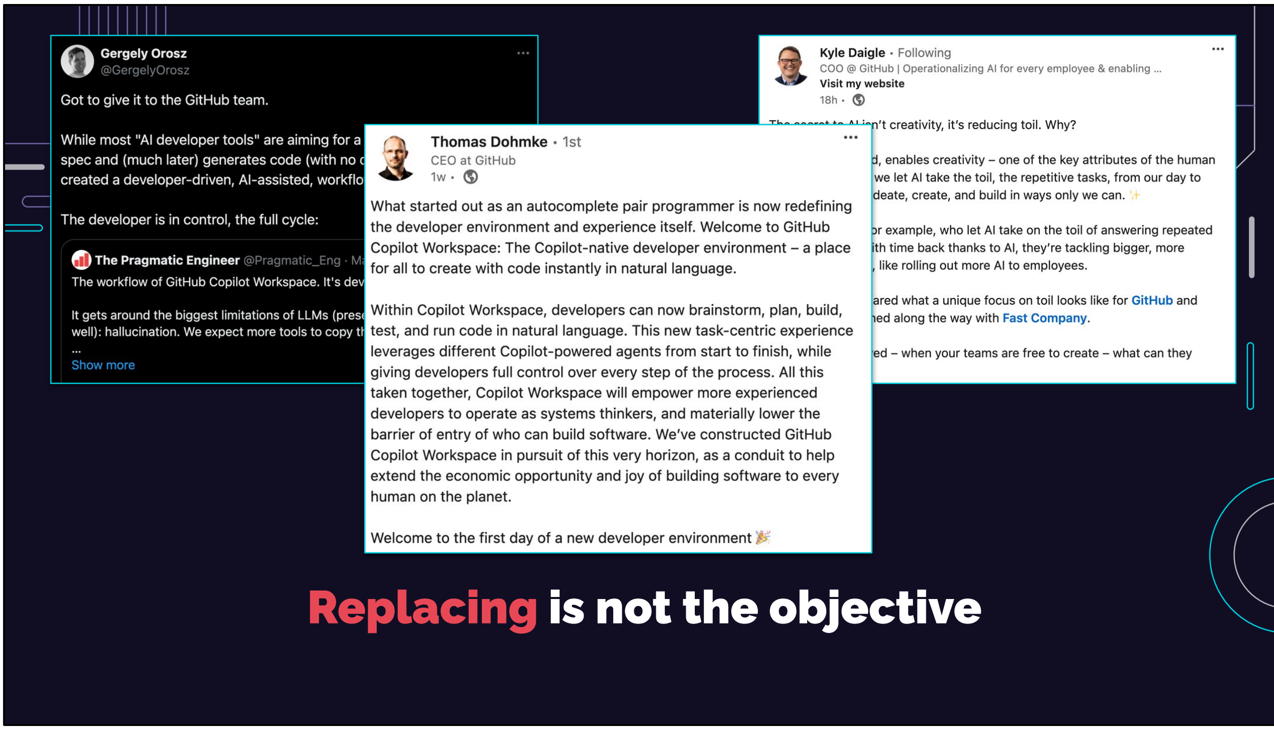
Toil, when removed, enables creativity – one of the key attributes of the human experience. When we let AI take the toil, the repetitive tasks, from our day to day, we're free to ideate, create, and build in ways only we can. ✨

Take our IT team for example, who let AI take on the toil of answering repeated questions. Now, with time back thanks to AI, they're tackling bigger, more valuable problems, like rolling out more AI to employees.

So glad to have shared what a unique focus on toil looks like for **GitHub** and insights we've gained along the way with **Fast Company**.

When toil is removed – when your teams are free to create – what can they accomplish? 💡

Replacing is not the objective


Gergely Orosz
@GergelyOrosz
Got to give it to the GitHub team.
While most "AI developer tools" are aiming for a
spec and (much later) generates code (with no c
created a developer-driven, AI-assisted, workflo
The developer is in control, the full cycle:
The Pragmatic Engineer @Pragmatic_Eng · M
The workflow of GitHub Copilot Workspace. It's dev
It gets around the biggest limitations of LLMs (prese
well): hallucination. We expect more tools to copy th
...
Show more
Kyle Daigle · Following
COO @ GitHub | Operationalizing AI for every employee & enabling ...
Visit my website
18h ·
The secret to AI isn't creativity, it's reducing toil. Why?
d, enables creativity – one of the key attributes of the human
we let AI take the toil, the repetitive tasks, from our day to
deate, create, and build in ways only we can.
or example, who let AI take on the toil of answering repeated
ith time back thanks to AI, they're tackling bigger, more
, like rolling out more AI to employees.
ared what a unique focus on toil looks like for GitHub and
ned along the way with Fast Company.
ed – when your teams are free to create – what can they
Thomas Dohmke · 1st
CEO at GitHub
1w ·
What started out as an autocomplete pair programmer is now redefining the developer environment and experience itself. Welcome to GitHub Copilot Workspace: The Copilot-native developer environment – a place for all to create with code instantly in natural language.
Within Copilot Workspace, developers can now brainstorm, plan, build, test, and run code in natural language. This new task-centric experience leverages different Copilot-powered agents from start to finish, while giving developers full control over every step of the process. All this taken together, Copilot Workspace will empower more experienced developers to operate as systems thinkers, and materially lower the barrier of entry of who can build software. We've constructed GitHub Copilot Workspace in pursuit of this very horizon, as a conduit to help extend the economic opportunity and joy of building software to every human on the planet.
Welcome to the first day of a new developer environment 🎉
Replacing is not the objective

This is the third premise.

The conclusion

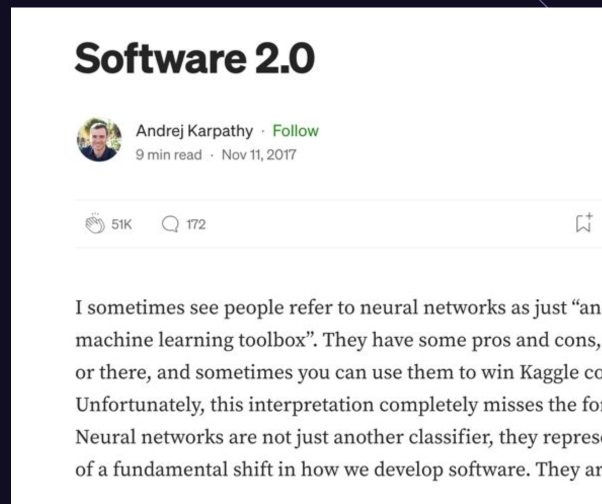
5 to 10 year horizon

In 2017 Andrej Karpathy wrote about **Software 2.0**

"Software 2.0 most often the source code comprises

1) the dataset that defines the desirable behavior and

2) the neural net architecture that gives the rough skeleton of the code, but with many details (the weights) to be filled in."



Software 2.0

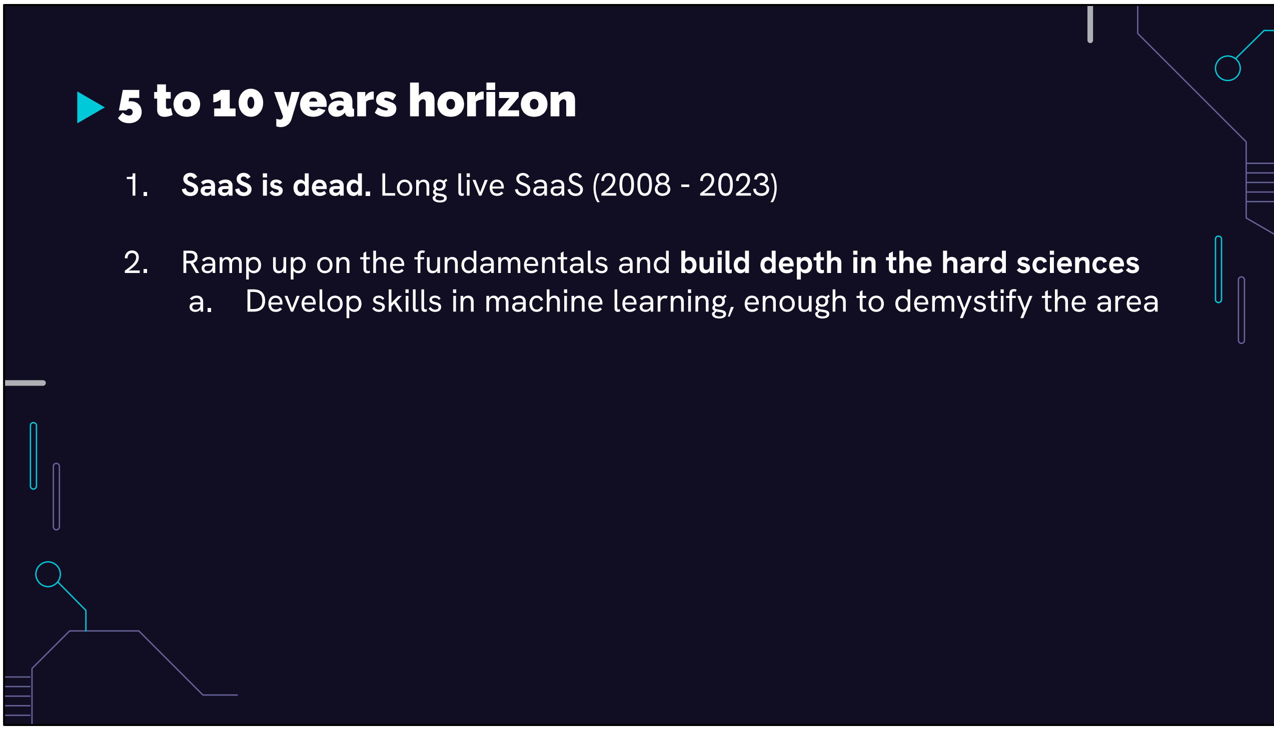
Andrej Karpathy · Follow
9 min read · Nov 11, 2017

51K 172

I sometimes see people refer to neural networks as just "ano
machine learning toolbox". They have some pros and cons, t
or there, and sometimes you can use them to win Kaggle co
Unfortunately, this interpretation completely misses the for
Neural networks are not just another classifier, they represe
of a fundamental shift in how we develop software. They are

5 to 10 years horizon

1. Change will be **gradual and expected**
2. LLMs **are not "it"**
   a. Objective-driven architectures will start to emerge (ability to plan, steer and converge)
3. **Human-level AI beyond this time horizon**
   a. But we will steadily make progress towards it
4. Stop the urge to **compete with the machine**
5. **AI assisted development will get better** and better



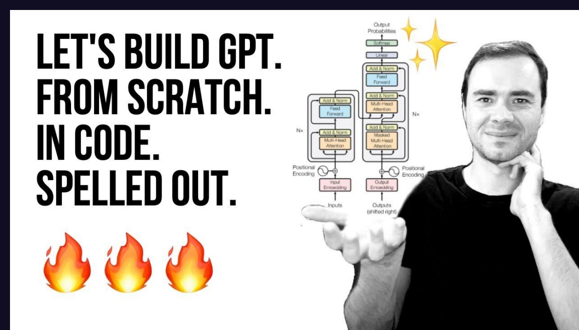
SaaS is a solved problem. We know well how to build and scale web based systems.

This era has new problems to solve.

5 to 10 years horizon

1. **SaaS is dead.** Long live SaaS (2008 - 2023)

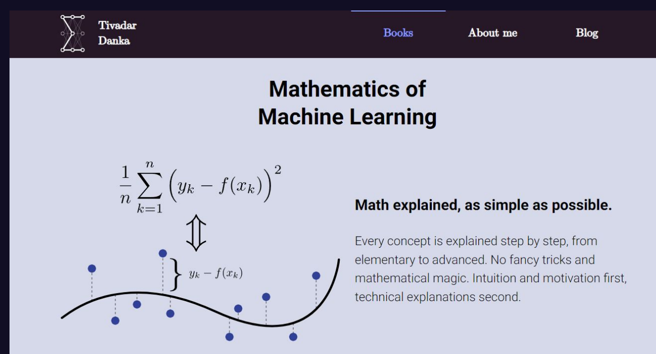
2. Ramp up on the fundamentals and **build depth in the hard sciences**
   a. Develop skills in machine learning, enough to demystify the area



5 to 10 years horizon

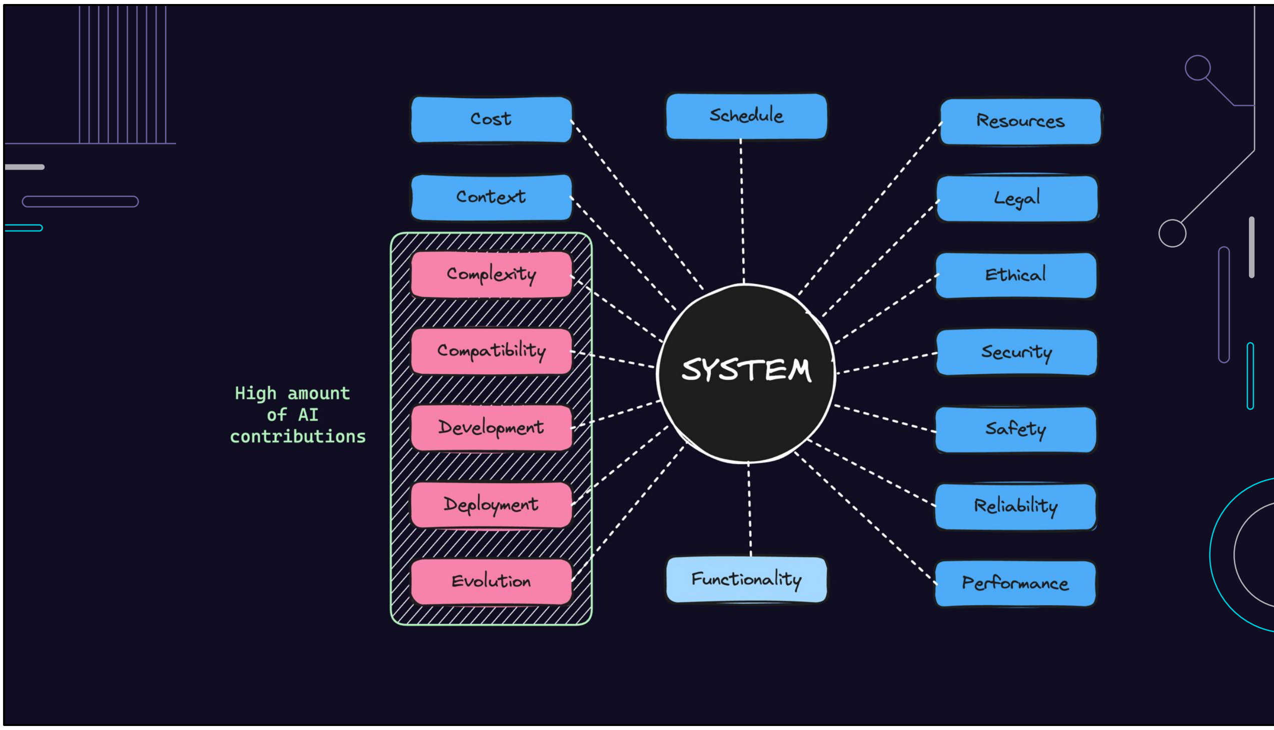
1. **SaaS is dead.** Long live SaaS (2008 - 2023)

2. Ramp up on the fundamentals and **build depth in the hard sciences**
   a. Develop skills in machine learning, enough to demystify the area



5 to 10 years horizon

1. **SaaS is dead.** Long live SaaS (2008 - 2023)
2. Ramp up on the fundamentals and **build depth in the hard sciences**
   a. Develop skills in machine learning, enough to demystify the area
3. **Less focus on syntactic sugar** and more focus on what's happening under the hood.
4. Programming will go back to being a hobby for many of us.
5. The spotlight shining on **solving "human" problems** will only get brighter. It's not about the tech, it never was.


Cost
Schedule
Resources
Context
Legal
Ethical
High amount
of AI
contributions
Complexity
Compatibility
Development
Deployment
Evolution
SYSTEM
Security
Safety
Reliability
Functionality
Performance

10+ years horizon

Nobody can see past the singularity.

In many ways, we will discover the future

But right now, we have a **responsibility to try and create a better one together**

Thank you!

SCAN ME

All the links you need and contact information:

https://glich.stream