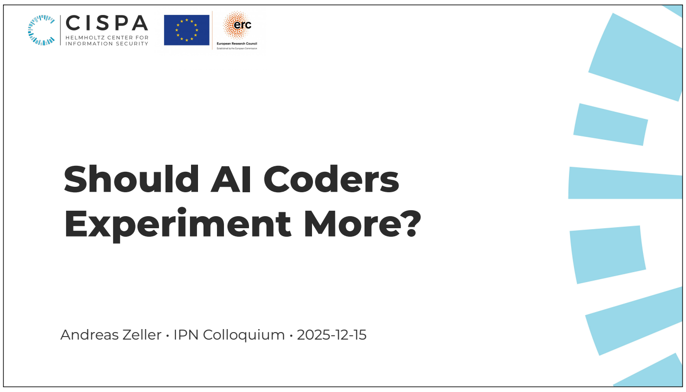


IPN Colloquium 6: Should AI Coders Experiment More?

Professor Andreas Zeller, CISPA Helmholtz Center for Information Security (DE)

To celebrate the achievements of IPN in the past 25 years, IPN is organising a special, online series of colloquia in which world-renowned computer scientists give their view on the progress in, and future of the field of computer science. These colloquia will feature thought-provoking presentations that are of interest to a broad (academic) computer science audience. Although these colloquia are initially aimed at the Dutch computer science community, they are open to interested people around the world!

Abstract

Most of today's AI coding assistance is based on the principles of Large Language Models (LLMs) – learning common token sequences from millions of annotated code examples, and then adapting them into the desired context. However, this "observation-only" approach has two problems: First, the reservoir of available code to learn from is limited; second, reasoning from abstract code to concrete executions is not a task LLMs have shown to excel at. In this talk, I sketch how future AI systems will be able to massively experiment with programs, their inputs, and their code in order to automatically learn how these programs behave, to predict the effects of their inputs and code changes, and in return predict and suggest actions on how to achieve arbitrary effects. Such AI systems will act as artificial program experts, tirelessly accumulating knowledge about the code and its environment, and – in contrast to current AI coders – be perceived as "super coders" that may become way more competent than the most experienced programmers: "Is there an input that bypasses authorization, and which is it?"

Andreas Zeller is faculty at the CISPA Helmholtz Center for Information Security and professor for Software Engineering at Saarland University. His research on automated debugging, mining software archives, specification mining, and security testing has proven highly influential. Zeller is one of the few researchers to have received two ERC Advanced Grants, most recently for his S3 project. Zeller is an ACM Fellow and holds an ACM SIGSOFT Outstanding Research Award.

Date, Time, Location

The colloquium will take place on December 15, 2025, 16:00 – 17:00 (CEST).

The colloquium will be hosted as a Teams webinar. You can join the colloquium via teams.

We are looking forward to seeing you there!

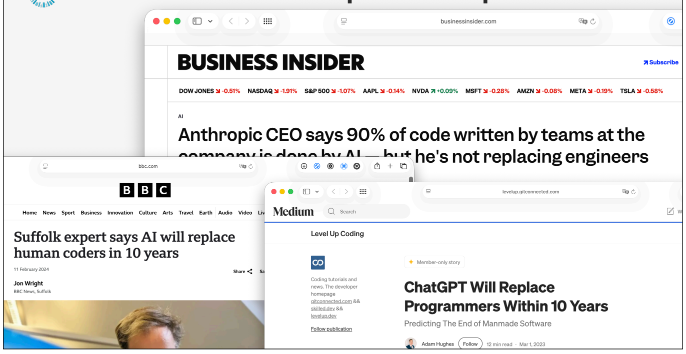
Afterwards you can find the recording and other recordings on the IPN colloquia overview page.

About me

Andreas Zeller is faculty at the CISPA Helmholtz Center for Information Security and professor for Software Engineering at Saarland University. His research on automated debugging, mining software archives, specification mining, and security testing has proven highly influential. Zeller is one of the few researchers to have received two ERC Advanced Grants, most recently for his S3 project. Zeller is an ACM Fellow and holds an ACM SIGSOFT Outstanding Research Award.

andreas.zeller@cispa.de
+49 681 87083-1001
@andreaszeller.bsky.social
@AndreasZeller@mastodon.social
andreaszeller
andreas-zeller

Will AI-Generated Code Replace Developers?



Predictions from 2024 and 2025: Will AI-Generated Code Replace Developers?

Will AI-Generated Code Replace Developers?

82% of developers use AI tools weekly

25% of new Google code is AI-assisted

Developers save time **30-60%** on coding, testing, docs

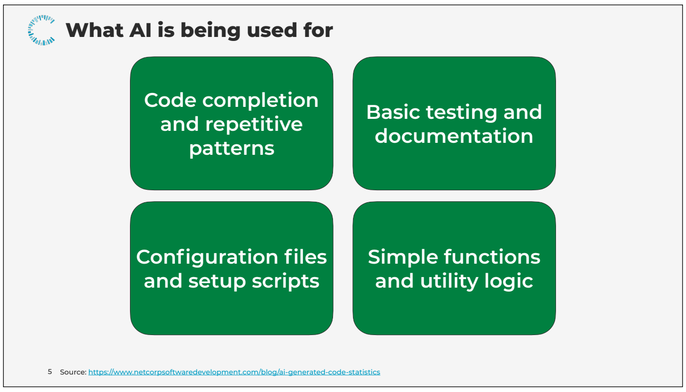
Code duplication is up **4x** with AI-generated code

75% of developers review every AI snippet

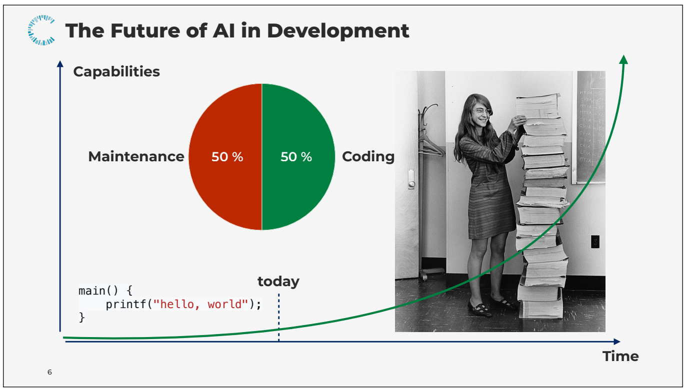
62% of developers are worried about technical debt

 Source: https://www.netcorpsoftwaredevelopment.com/blog/ai-generated-code-statistics

Current statistics of 2025: Yes, AI is everywhere – but also creates problems

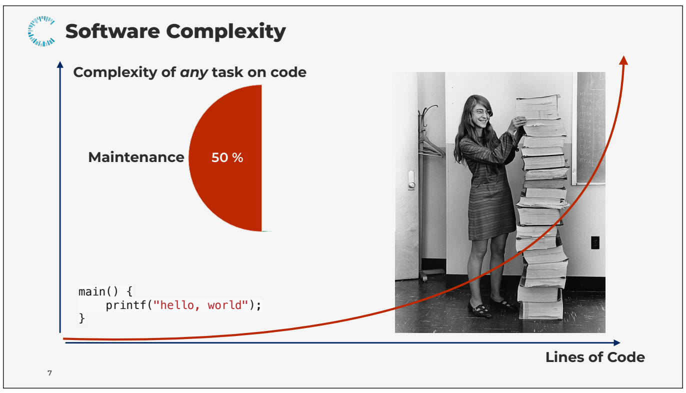


The things AI can do are nice and useful. But they are still solving relatively simple tasks..

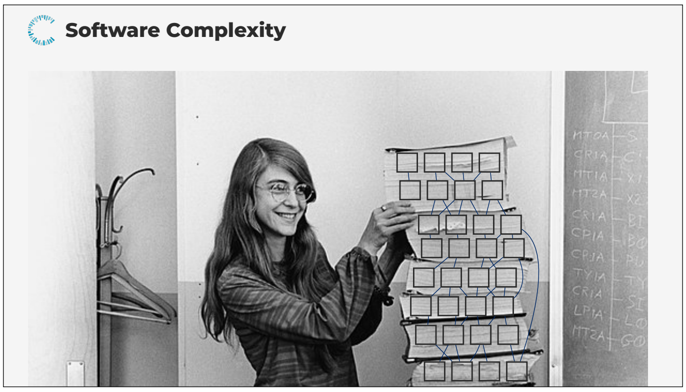


The assumption is that AI will get so much better it can handle even the largest coding problems. Software Engineering is 50% coding and 50% maintenance of existing code. Can we expect AI also to _maintain_ existing systems? To _integrate_ with them?

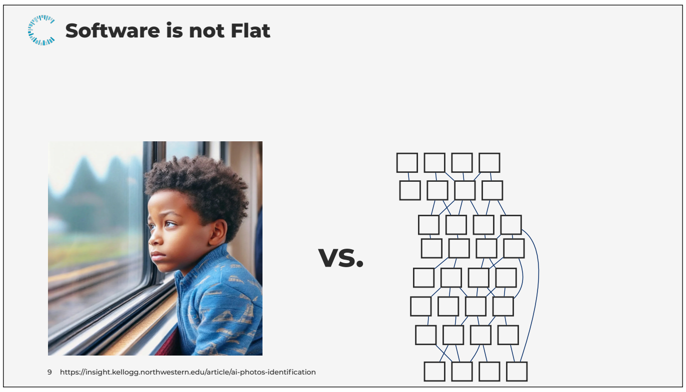
The NASA photo shows Margret Hamilton with a printout of the Apollo 11 source code. Can we trust AI to produce code of this size and criticality?



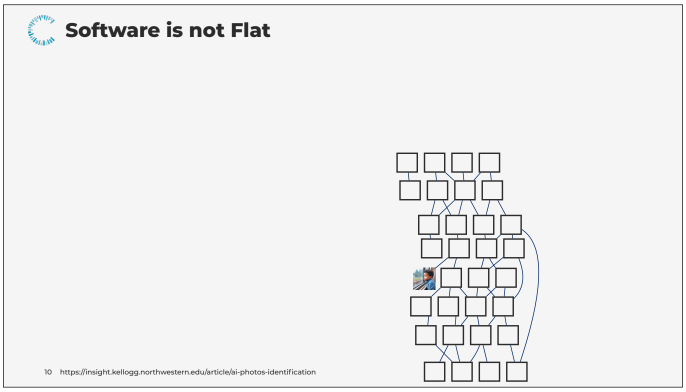
The problem is, however, that any task on code (writing, understanding) also has exponential growth the larger the program becomes.



The problem is that we have millions of cross-references between code entities. Everything is related to everything else – and can break anything.

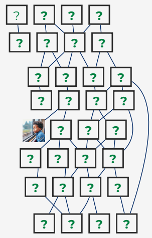


This is in sharp contrast to (LLM-generated) images, which have far fewer cross-references



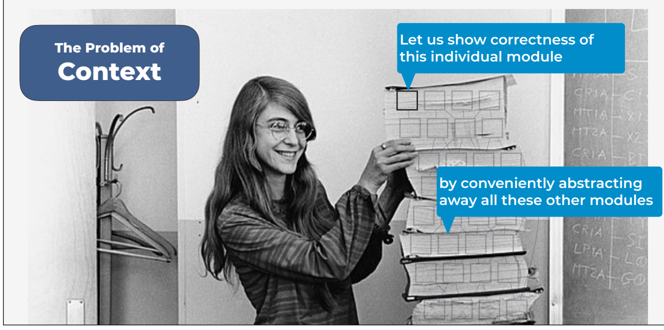
While AI can easily create a single picture, maintaining consistency across multiple pictures or sceneries is a big challenge (aka unsolved problem). This problem of consistency is the same in software.

Software is not Flat



11 https://insight.kellogg.northwestern.edu/article/ai-photos-identification

The Problem of Context



We could go and examine one module after another, reducing scale and context. This is helpful for establishing trust in a single component (say, as in symbolic verification) in itself, but not necessarily in the entire context.

The Problem of Context

```
# Copyright: Public domain.
# Filename:  BURN_BABY_BURN--MASTER_IGNITION_ROUTINE.agc
# Purpose:   Part of the source code for Luminary 1A build 099.
#     It is part of the source code for the Lunar Module's (LM)
#     Apollo Guidance Computer (AGC), for Apollo 11.
# Assembler: yaYUL
# Contact:   Ron Burkey <info@sandroid.org>.
# Website:   www.ibiblio.org/apollo.
# Pages:  731-751
# Mod history:  2009-05-19 RSB  Adapted from the corresponding
#           Luminary131 file, using page
#           images from Luminary 1A.
#     2009-06-07 RSB  Corrected 3 typos.
#     2009-07-23 RSB  Added Onno's notes on the naming
#           of this function, which he got from
#           Don Eyles.
#
# This source code has been transcribed or otherwise adapted from
# digitized images of a hardcopy from the MIT Museum.  The digitization
# was performed by Pau
```

13 Source: https://github.com/chrislgarry/Apollo-11/blob/master/Luminary099/BURN_BABY_BURN--MASTER_IGNITION_ROUTINE.agc

Context is the central problem here. This is an excerpt of the Apollo 11 code. How can anyone (or anything) understand what is going on here without knowing the context of things? Without the engineering documentation? Without the actual engineers? Without the history and motivation?

The Problem of Context

```
## naming of the routine.
##
## It traces back to 1965 and the Los Angeles riots, and was inspired
## by disc jockey extraordinaire and radio station owner Magnificent Montague.
## Magnificent Montague used the phrase "Burn, baby! BURN!" when spinning the
## hottest new records. Magnificent Montague was the charismatic voice of
## soul music in Chicago, New York, and Los Angeles from the mid-1950s to
## the mid-1960s.
# BURN, BABY, BURN -- MASTER IGNITION ROUTINE

      BANK   36
      SETLOC P40S
      BANK
      EBANK= WHICH
      COUNT* $$/P40

# THE MASTER IGNITION ROUTINE IS DESIGNED FOR USE BY THE FOLLOWING LEM PROGRAMS:
P12, P40, P42, P61, P63.
# IT PERFORMS ALL FUNCTIONS IMMEDIATELY ASSOCIATED WITH APS OR DPS IGNITION:  IN
PARTICULAR, EVERYTHING LYING
# BETWEEN THE PRE-IGNITION TIME CHECK -- ARE WE WITHIN 45 SECONDS OF TIG? -- AND
TIG + 26 SECONDS, WHEN DPS
# PROGRAMS THROTTLE UP.
```

What is this code doing? You'd need detailed context about the CPU used, the libraries used, the construction diagrams of the lunar lander, and much much more.

The Problem of Context

```
#      HONI SOIT QUI MAL Y PENSE
#
#  ************************************************
#     TABLES FOR THE IGNITION ROUTINE
#  ************************************************
#
#        NOLI SE TANGERE

P12TABLE  VN 0674      # (0)
      TCF ULLGNOT      # (1)
      TCF COMFAIL3  # (2)
      TCF GOCUTOFF  # (3)
      TCF TASKOVER  # (4)
      TCF P12SPOT      # (5)
      DEC 0     # (6) NO ULLAGE
      EBANK= WHICH
      2CADR  SERVEXIT  # (7)

      TCF DISPCHNG  # (11)
      TCF WAITABIT  # (12)
      TCF P12IGN    # (13)

P40TABLE  VN 0640      # (0)
```

What is this code doing?

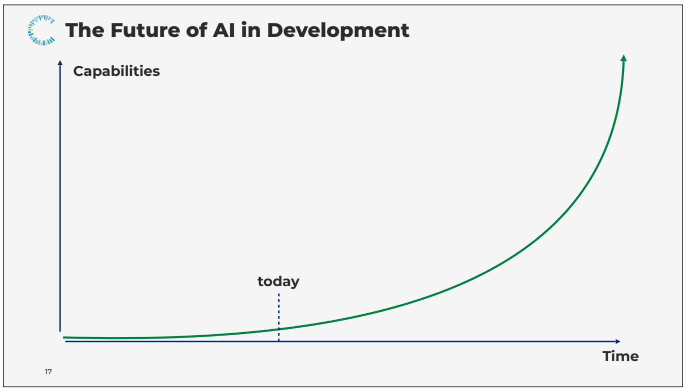
Noli se tangere = do not touch

The Problem of Context

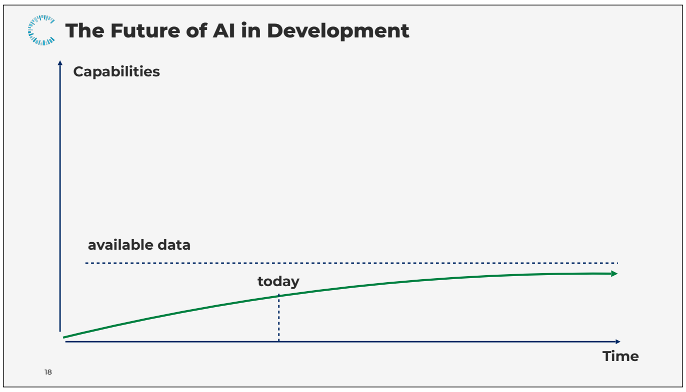
```
#  ***********************************
ONULLAGE  CS DAPBOOLS  # TURN ON ULLAGE.  MUST BE CALLED IN
      MASK  ULLAGER      # A TASK OR WHILE INHINTED.
      ADS DAPBOOLS
      TC Q

#  ***********************************

STCLOK1      CA  ZERO      # THIS ROUTINE STARTS THE COUNT-DOWN
STCLOK2      TS  DISPDEX      # (CLOKTASK AND CLOKJOB).  SETTING
STCLOK3      TC  MAKECADR  # SETTING DISPDEX POSITIVE KILLS IT.
      TS  TBASE4    # RETURN SAVE (NOT FOR RESTARTS).
      EXTEND
      DCA TIG
      DXCH  MPAC
      EXTEND
      DCS TIME2
# Page 744
      DAS MPAC      # HAVE TIG -- TIME2, UNDOUBTEDLY A + NUMBER
      TC  TPAGREE      # POSITIVE, SINCE WE PASSED THE
      CAF 1SEC      # 45 SECOND CHECK.
      TS  Q
```

This is the problem of context: Out of context, you won't be able to understand

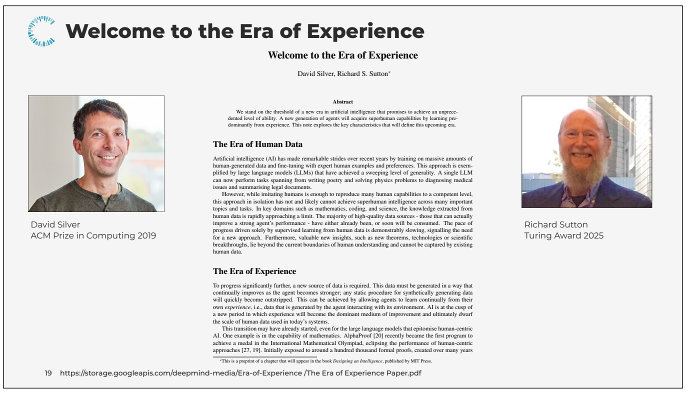
At its height, Nasa estimates that a total of 400,000 men and women across the United States were involved in the Apollo programme



The lack of context actually translates into a larger problem: In general terms, AI might suffer from a _lack of data_ to train from



Which would lead to this maybe more realistic projection of future AI capabilities



Welcome to the Era of Experience

David Silver, Richard S. Sutton*

Abstract

We stand on the threshold of a new era in artificial intelligence that promises to achieve an unprecedented level of ability. A new generation of agents will acquire superhuman capabilities by learning predominantly from experience. This note explores the key characteristics that will define this upcoming era.

The Era of Human Data

Artificial intelligence (AI) has made remarkable strides over recent years by training on massive amounts of human-generated data and fine-tuning with expert human examples and preferences. This approach is exemplified by large language models (LLMs) that have achieved a sweeping level of generality. A single LLM can now perform tasks spanning from writing poetry and solving physics problems to diagnosing medical issues and summarising legal documents.

However, while imitating humans is enough to reproduce many human capabilities to a competent level, this approach in isolation has not and likely cannot achieve superhuman intelligence across many important topics and tasks. In key domains such as mathematics, coding, and science, the knowledge extracted from human data is rapidly approaching a limit. The majority of high-quality data sources - those that can actually improve a strong agent's performance - have either already been, or soon will be consumed. The pace of progress driven solely by supervised learning from human data is demonstrably slowing, signalling the need for a new approach. Furthermore, valuable new insights, such as new theorems, technologies or scientific breakthroughs, lie beyond the current boundaries of human understanding and cannot be captured by existing human data.

The Era of Experience

To progress significantly further, a new source of data is required. This data must be generated in a way that continually improves as the agent becomes stronger; any static procedure for synthetically generating data will quickly become outstripped. This can be achieved by allowing agents to learn continually from their own *experience*, i.e., data that is generated by the agent interacting with its environment. AI is at the cusp of a new period in which experience will become the dominant medium of improvement and ultimately dwarf the scale of human data used in today's systems.

This transition may have already started, even for the large language models that epitomise human-centric AI. One example is in the capability of mathematics. AlphaProof [20] recently became the first program to achieve a medal in the International Mathematical Olympiad, eclipsing the performance of human-centric approaches [27, 19]. Initially exposed to around a hundred thousand formal proofs, created over many years

*This is a preprint of a chapter that will appear in the book *Designing an Intelligence*, published by MIT Press.

This was also anticipated earlier this year in this beautiful essay

The Era of Human Data

Artificial intelligence (AI) has made remarkable strides over recent years by training on massive amounts of human-generated data and fine-tuning with expert human examples and preferences. This approach is exemplified by large language models (LLMs) that have achieved a sweeping level of generality. A single LLM can now perform tasks spanning from writing poetry and solving physics problems to diagnosing medical issues and summarising legal documents.

However, while imitating humans is enough to reproduce many human capabilities to a competent level, this approach in isolation has not and likely cannot achieve superhuman intelligence across many important topics and tasks. In key domains such as mathematics, coding, and science, the knowledge extracted from human data is rapidly approaching a limit. The majority of high-quality data sources - those that can actually improve a strong agent's performance - have either already been, or soon will be consumed. The pace of progress driven solely by supervised learning from human data is demonstrably slowing, signalling the need for a new approach. Furthermore, valuable new insights, such as new theorems, technologies or scientific breakthroughs, lie beyond the current boundaries of human understanding and cannot be captured by existing human data.

predicting that we lack sufficient data to train further AI systems

---

The Era of Experience

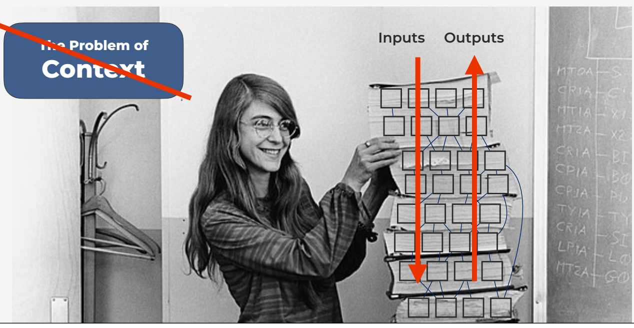
To progress significantly further, a new source of data is required. This data must be generated in a way that continually improves as the agent becomes stronger; any static procedure for synthetically generating data will quickly become outstripped. This can be achieved by allowing agents to learn continually from their own *experience*, i.e., data that is generated by the agent interacting with its environment. AI is at the cusp of a new period in which experience will become the dominant medium of improvement and ultimately dwarf the scale of human data used in today's systems.

This transition may have already started, even for the large language models that epitomise human-centric AI. One example is in the capability of mathematics. AlphaProof [20] recently became the first program to achieve a medal in the International Mathematical Olympiad, eclipsing the performance of human-centric approaches [27, 19]. Initially exposed to around a hundred thousand formal proofs, created over many years

and therefore suggesting that AI should _experiment more_ to learn from its experience

---

Experimenting with Software

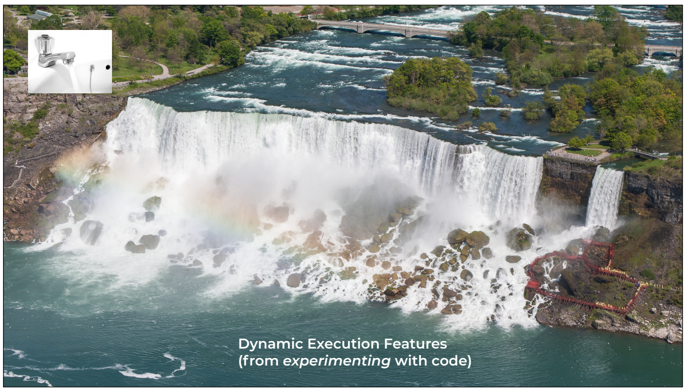


Good news: With software, we can experiment at will, without getting into problems of abstraction, scale, and context

Static Code Features
(from *observing* code)

23

The thing is: If you learn from code, there's only so much you can infer (in particular, if you largely ignore or cannot reason about the _semantics_ of code)



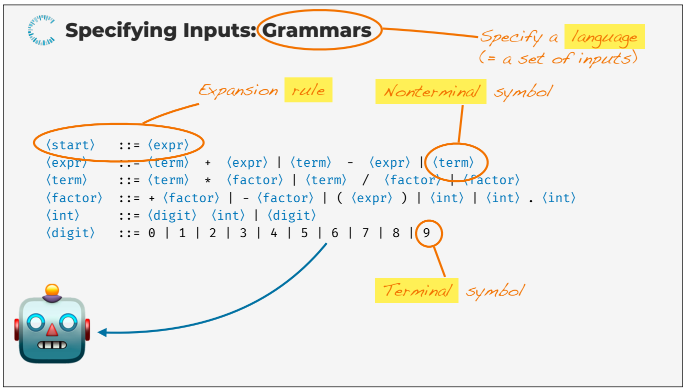
But if you look at the data that becomes available during _execution_, then a _wealth_ of data becomes available. That is the power of experimentation.

Experimenting with Software

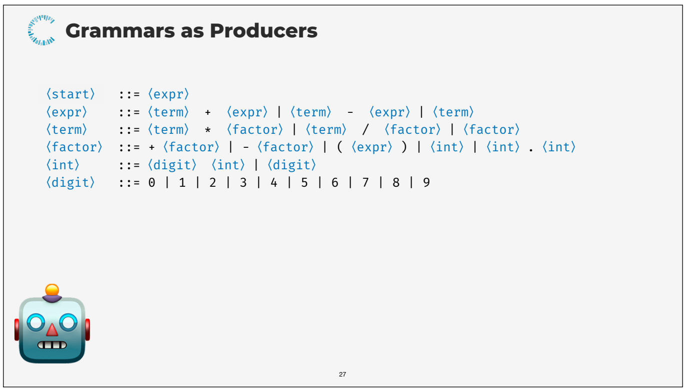
- Where do we get inputs from?
- How do we process outputs?
- How do we train an AI with that?

25 Source: https://www.netcorpsoftwaredevelopment.com/blog/ai-generated-code-statistics

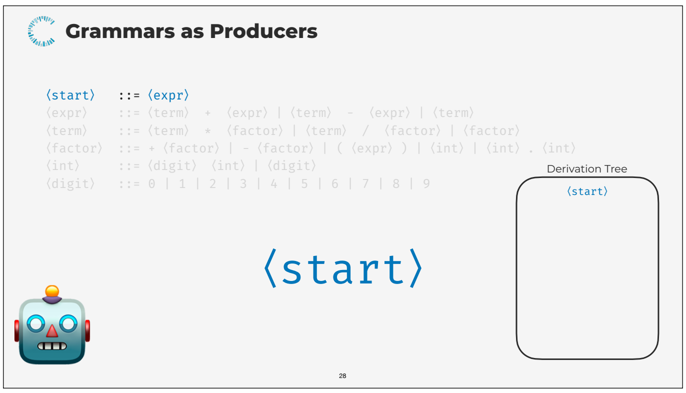
So these are our challenges. Good news: As a software engineer, I have solutions for all these



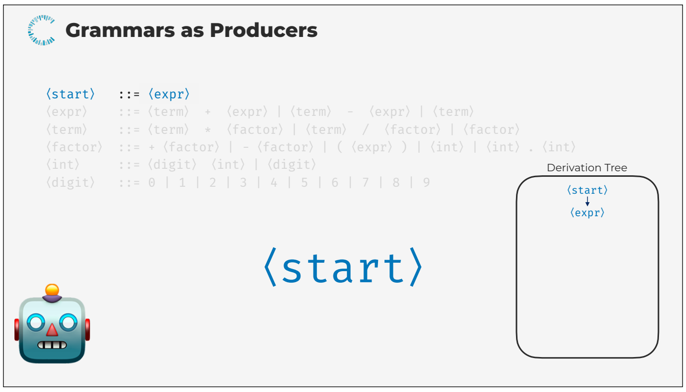
Let's get into the first problem – how to get input data. A classical approach is to _specify_ the inputs we need – say, with a grammar.

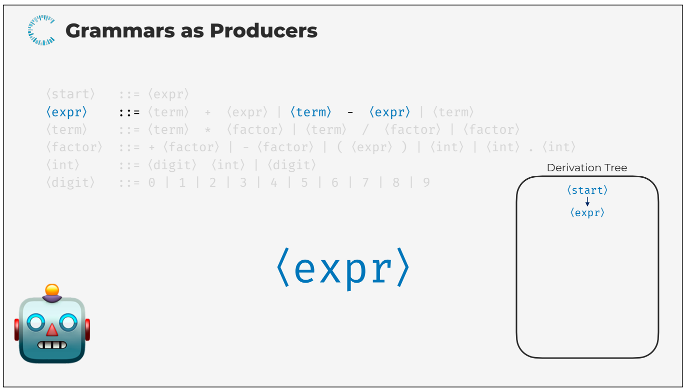


You may have seen grammars as **parsers**, but they can also be used as **producers** of inputs.

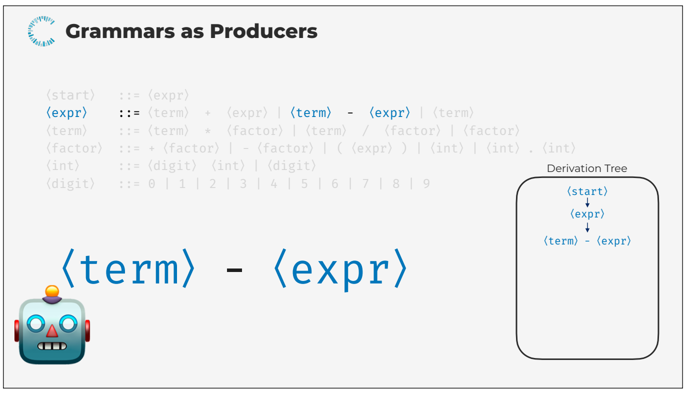


You start with a start symbol





which then subsequently gets replaced according to the production rules in the grammar.



If there are multiple alternatives, you randomly choose one.

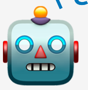
Grammars as Producers

```
⟨start⟩   ::= ⟨expr⟩
⟨expr⟩    ::= ⟨term⟩ + ⟨expr⟩ | ⟨term⟩ - ⟨expr⟩ | ⟨term⟩
⟨term⟩    ::= ⟨term⟩ * ⟨factor⟩ | ⟨term⟩ / ⟨factor⟩ | ⟨factor⟩
⟨factor⟩  ::= + ⟨factor⟩ | - ⟨factor⟩ | ( ⟨expr⟩ ) | ⟨int⟩ | ⟨int⟩ . ⟨int⟩
⟨int⟩     ::= ⟨digit⟩ ⟨int⟩ | ⟨digit⟩
⟨digit⟩   ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Derivation Tree

```
⟨start⟩
   ↓
⟨expr⟩
   ↓
⟨term⟩ - ⟨expr⟩
```

⟨term⟩ - ⟨expr⟩



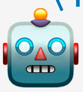
Grammars as Producers

```
⟨start⟩   ::= ⟨expr⟩
⟨expr⟩    ::= ⟨term⟩ + ⟨expr⟩ | ⟨term⟩ - ⟨expr⟩ | ⟨term⟩
⟨term⟩    ::= ⟨term⟩ * ⟨factor⟩ | ⟨term⟩ / ⟨factor⟩ | ⟨factor⟩
⟨factor⟩  ::= + ⟨factor⟩ | - ⟨factor⟩ | ( ⟨expr⟩ ) | ⟨int⟩ | ⟨int⟩ . ⟨int⟩
⟨int⟩     ::= ⟨digit⟩ ⟨int⟩ | ⟨digit⟩
⟨digit⟩   ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Derivation Tree

```
⟨start⟩
   ↓
⟨expr⟩
   ↓
⟨term⟩ - ⟨expr⟩
   ↓         ↓
⟨factor⟩     …
```

⟨factor⟩ - ⟨expr⟩



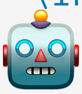
Grammars as Producers

```
⟨start⟩   ::= ⟨expr⟩
⟨expr⟩    ::= ⟨term⟩ + ⟨expr⟩ | ⟨term⟩ - ⟨expr⟩ | ⟨term⟩
⟨term⟩    ::= ⟨term⟩ * ⟨factor⟩ | ⟨term⟩ / ⟨factor⟩ | ⟨factor⟩
⟨factor⟩  ::= + ⟨factor⟩ | - ⟨factor⟩ | ( ⟨expr⟩ ) | ⟨int⟩ | ⟨int⟩ . ⟨int⟩
⟨int⟩     ::= ⟨digit⟩ ⟨int⟩ | ⟨digit⟩
⟨digit⟩   ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Derivation Tree

```
⟨start⟩
   ↓
⟨expr⟩
   ↓
⟨term⟩ - ⟨expr⟩
   ↓         ↓
⟨factor⟩     …
   ↓
⟨int⟩ . ⟨int⟩
```

⟨int⟩ . ⟨int⟩ - ⟨expr⟩



Grammars as Producers

```
⟨start⟩   ::= ⟨expr⟩
⟨expr⟩    ::= ⟨term⟩  +  ⟨expr⟩ | ⟨term⟩  -  ⟨expr⟩ | ⟨term⟩
⟨term⟩    ::= ⟨term⟩  *  ⟨factor⟩ | ⟨term⟩  /  ⟨factor⟩ | ⟨factor⟩
⟨factor⟩  ::= + ⟨factor⟩ | - ⟨factor⟩ | ( ⟨expr⟩ ) | ⟨int⟩ | ⟨int⟩ . ⟨int⟩
⟨int⟩     ::= ⟨digit⟩  ⟨int⟩ | ⟨digit⟩
⟨digit⟩   ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

⟨digit⟩ . ⟨int⟩ - ⟨expr⟩

Derivation Tree

```
⟨start⟩
⟨expr⟩
⟨term⟩ - ⟨expr⟩
⟨factor⟩    …
⟨int⟩ . ⟨int⟩
⟨digit⟩
```

Grammars as Producers

```
⟨start⟩   ::= ⟨expr⟩
⟨expr⟩    ::= ⟨term⟩  +  ⟨expr⟩ | ⟨term⟩  -  ⟨expr⟩ | ⟨term⟩
⟨term⟩    ::= ⟨term⟩  *  ⟨factor⟩ | ⟨term⟩  /  ⟨factor⟩ | ⟨factor⟩
⟨factor⟩  ::= + ⟨factor⟩ | - ⟨factor⟩ | ( ⟨expr⟩ ) | ⟨int⟩ | ⟨int⟩ . ⟨int⟩
⟨int⟩     ::= ⟨digit⟩  ⟨int⟩ | ⟨digit⟩
⟨digit⟩   ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

⟨digit⟩ . ⟨digit⟩ - ⟨expr⟩

Derivation Tree

```
⟨start⟩
⟨expr⟩
⟨term⟩ - ⟨expr⟩
⟨factor⟩    …
⟨int⟩ . ⟨int⟩
⟨digit⟩ ⟨digit⟩
```

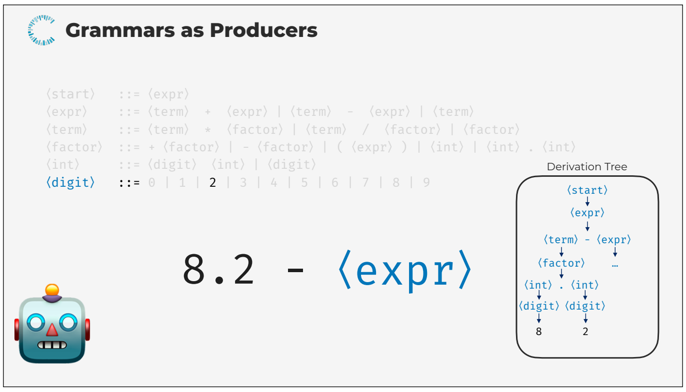
Grammars as Producers

```
⟨start⟩   ::= ⟨expr⟩
⟨expr⟩    ::= ⟨term⟩  +  ⟨expr⟩ | ⟨term⟩  -  ⟨expr⟩ | ⟨term⟩
⟨term⟩    ::= ⟨term⟩  *  ⟨factor⟩ | ⟨term⟩  /  ⟨factor⟩ | ⟨factor⟩
⟨factor⟩  ::= + ⟨factor⟩ | - ⟨factor⟩ | ( ⟨expr⟩ ) | ⟨int⟩ | ⟨int⟩ . ⟨int⟩
⟨int⟩     ::= ⟨digit⟩  ⟨int⟩ | ⟨digit⟩
⟨digit⟩   ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

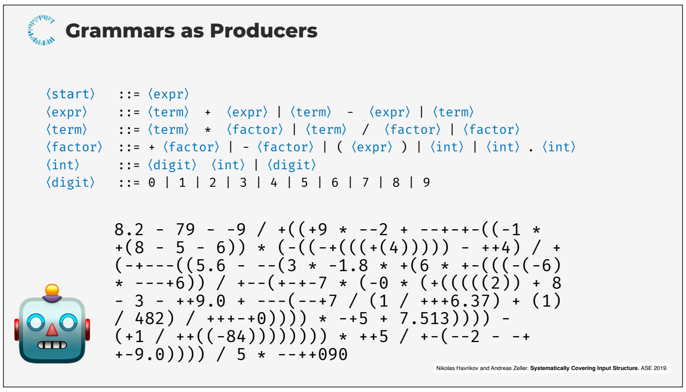
8. ⟨digit⟩ - ⟨expr⟩

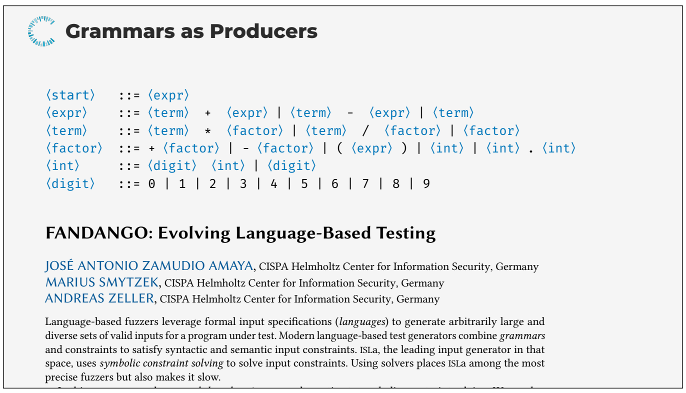
Derivation Tree

```
⟨start⟩
⟨expr⟩
⟨term⟩ - ⟨expr⟩
⟨factor⟩    …
⟨int⟩ . ⟨int⟩
⟨digit⟩ ⟨digit⟩
8
```



Over time, this gives you a syntactically valid input. In our case, a valid arithmetic expression.





In our own research work, we have developed very strong input generators (aka test generators, aka fuzzers) that combine such grammars with semantic properties (predicates over grammar elements) to express arbitrary features of the desired input.

Grammars as Producers

```
⟨start⟩   ::= ⟨expr⟩
⟨expr⟩    ::= ⟨term⟩  +  ⟨expr⟩ | ⟨term⟩  -  ⟨expr⟩ | ⟨term⟩
⟨term⟩    ::= ⟨term⟩  *  ⟨factor⟩ | ⟨term⟩  /  ⟨factor⟩ | ⟨factor⟩
⟨factor⟩  ::= + ⟨factor⟩ | - ⟨factor⟩ | ( ⟨expr⟩ ) | ⟨int⟩ | ⟨int⟩ . ⟨int⟩
⟨int⟩     ::= ⟨digit⟩  ⟨int⟩ | ⟨digit⟩
⟨digit⟩   ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Fandango

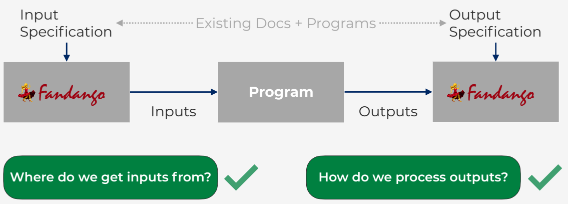
- Specifies inputs as **grammars** + (Python) **constraints**
- Solves **constraints** using **evolutionary algorithms**
- Expressive · modular · **fast**

https://fandango-fuzzer.github.io/

One popular tool of ours is called Fandango – it can produce and parse strings on demand

Testing Programs

We use Fandango to **test programs** – producing input and parsing output

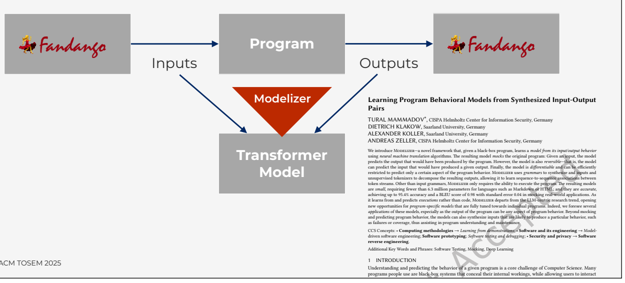


42

Thus, we know how to produce synthetic inputs and how to parse results

Training AI Models with Modelizer

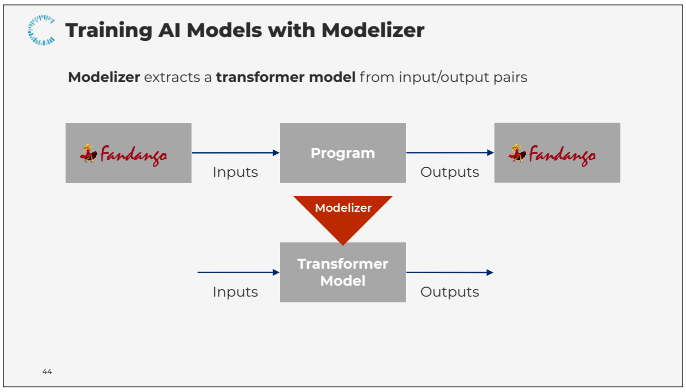
Modelizer extracts a **transformer model** from input/output pairs



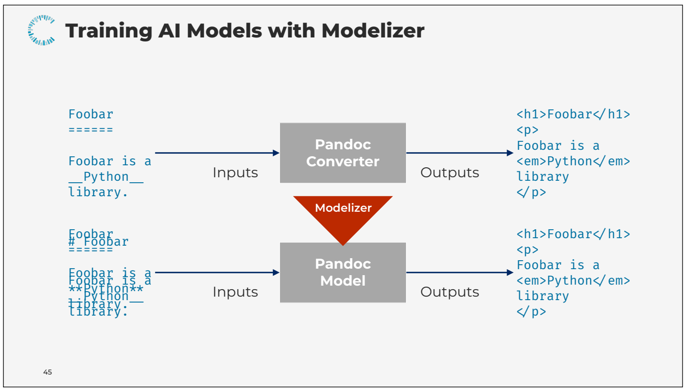
43 ACM TOSEM 2025

Based on these inputs and outputs (and their features), we can train _transformer_ models on input/output pairs.

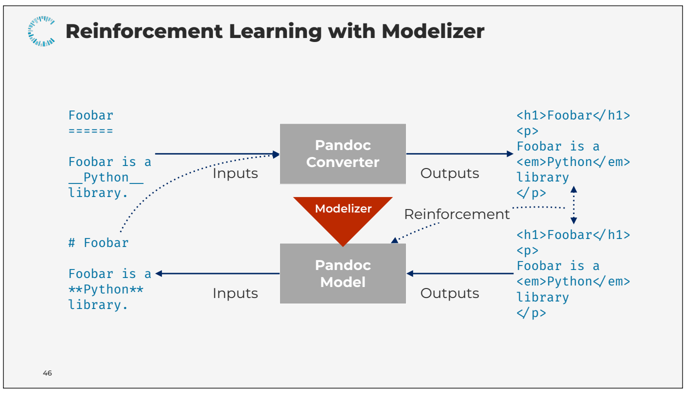
These are “vanilla” sequence-to-sequence neural machine translation models, as in “Attention is all you need”



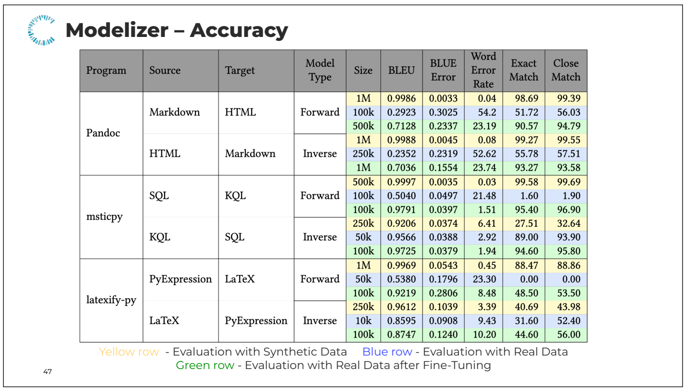
These models then _replicate_ the original behavior.



For instance, transforming Markdown to HTML can be exactly reproduced.

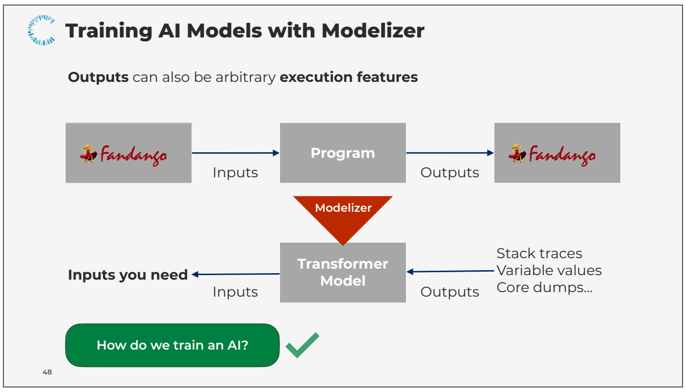


The nice thing, though, is that we can also predict _inputs_ from (desired) _outputs_! In other words, we feed in the behavior we'd like to see, and then the model predicts an input that produces this behavior.

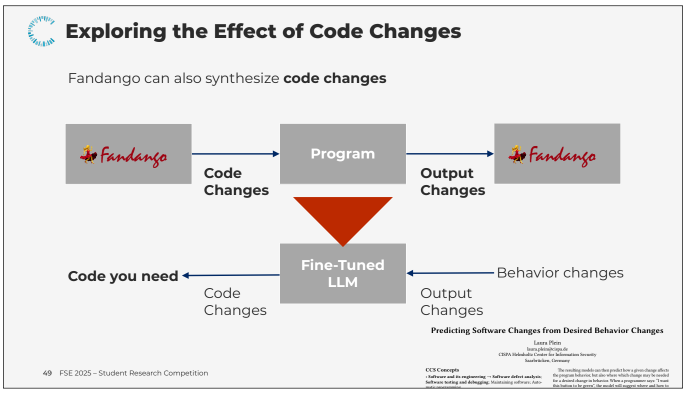


Program	Source	Target	Model Type	Size	BLEU	BLUE Error	Word Error Rate	Exact Match	Close Match
Pandoc	Markdown	HTML	Forward	1M	0.9986	0.0033	0.04	98.69	99.39
				100k	0.2923	0.3025	54.2	51.72	56.03
				500k	0.7128	0.2337	23.19	90.57	94.79
	HTML	Markdown	Inverse	1M	0.9988	0.0045	0.08	99.27	99.55
				250k	0.2352	0.2319	52.62	55.78	57.51
				1M	0.7036	0.1554	23.74	93.27	93.58
msticpy	SQL	KQL	Forward	500k	0.9997	0.0035	0.03	99.58	99.69
				100k	0.5040	0.0497	21.48	1.60	1.90
				100k	0.9791	0.0397	1.51	95.40	96.90
	KQL	SQL	Inverse	250k	0.9206	0.0374	6.41	27.51	32.64
				50k	0.9566	0.0388	2.92	89.00	93.90
				100k	0.9725	0.0379	1.94	94.60	95.80
latexify-py	PyExpression	LaTeX	Forward	1M	0.9969	0.0543	0.45	88.47	88.86
				50k	0.5380	0.1796	23.30	0.00	0.00
				100k	0.9219	0.2806	8.48	48.50	53.50
	LaTeX	PyExpression	Inverse	250k	0.9612	0.1039	3.39	40.69	43.98
				10k	0.8595	0.0908	9.43	31.60	52.40
				100k	0.8747	0.1240	10.20	44.60	56.00

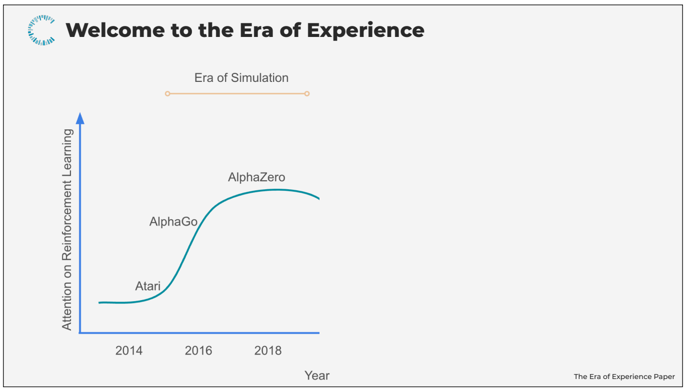
We get pretty accurate results with that; training takes less than 3 hours on my (old) laptop



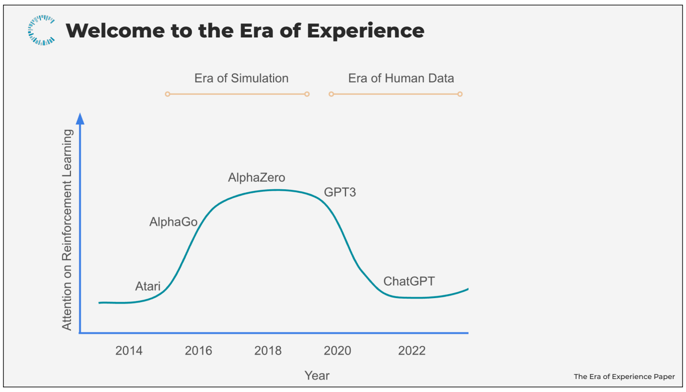
The "output" is actually anything we can observe during execution; so we can train an AI on all these execution features, and predict how to obtain them

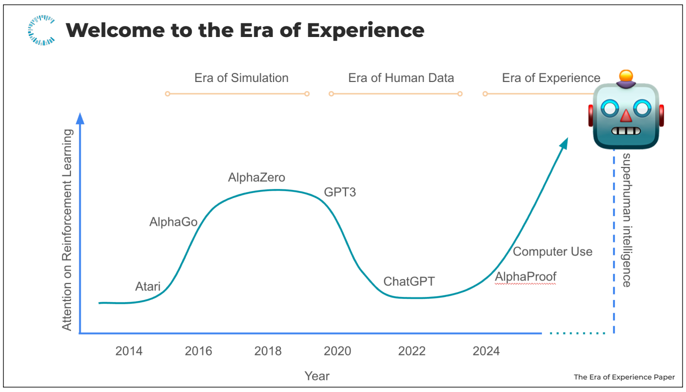


Another line of work studies the effect of _code changes_ on program output – and can again predict which code changes are required to obtain a particular effect in behavior

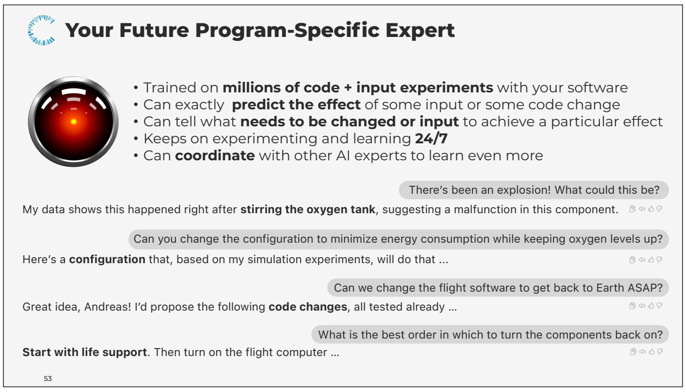


Now these are all baby steps towards a greater goal. Here's a graph from the era of experience paper: By allowing AI agents to experiment, we may eventually get towards super-intelligence

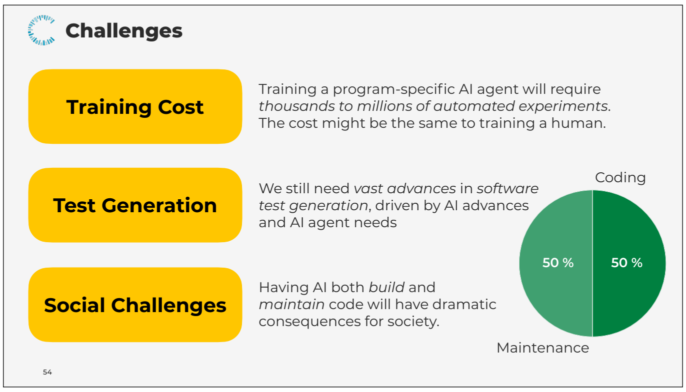




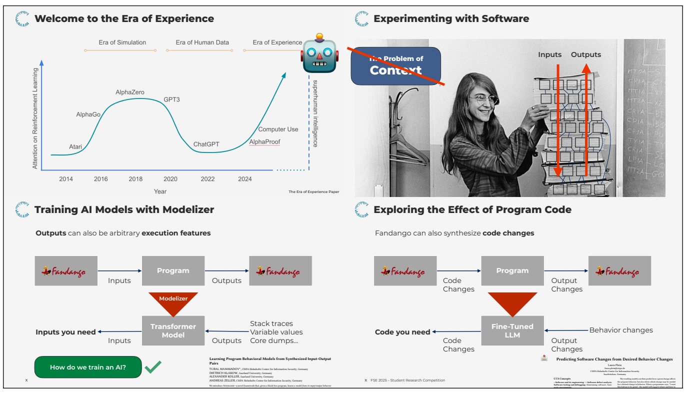
... which, in our case, might well be superhuman intelligence in program expertise.



This bodes well for future AI coding experts, trained on and for specific programs, and then being able to help with maintenance and debugging – at a scale where observation alone won't get you far. Interacting with such AI systems will actually feel like talking to a superhuman intelligence that already has seen and anticipated everything – from its experience.



Such "superhuman" AI agents would then address the second half of software engineering, namely maintaining and evolving existing systems. There are still several challenges on the road ahead, and this is good, because we need to prepare for the consequences for society.



And it's a wrap! Thanks a lot, and I am happy to take questions.