Validly

Modern-day code modernization

70% of large companies run on legacy infrastructure

```
IDENTIFICATION DIVISION.
       PROGRAM-ID. ADD_NUMBERS.
       DATA DIVISION.
       FILE SECTION.
       WORKING-STORAGE SECTION.
       01  FIRST-NUMBER    PICTURE IS 99.
       01  SECOND-NUMBER   PICTURE IS 99.
       01  RESULT          PICTURE IS 9999.
       PROCEDURE DIVISION.

       MAIN-PROCEDURE.
           DISPLAY "Here is the first Number "
           MOVE 8 TO FIRST-NUMBER
           DISPLAY FIRST-NUMBER

           DISPLAY "Let's add 20 to that number."
           ADD 20 TO FIRST-NUMBER
           DISPLAY FIRST-NUMBER

           DISPLAY "Create a second variable"
           MOVE 30 TO SECOND-NUMBER
           DISPLAY SECOND-NUMBER

           *>COMMENT: COMPUTE THE TWO NUMBER AND PLACE INTO RESULT*
           COMPUTE RESULT = FIRST-NUMBER +  SECOND-NUMBER.

           DISPLAY "The result is:".
           DISPLAY RESULT.
         STOP RUN.
END PROGRAM ADD_NUMBERS.
```

Companies spend billions modernizing

Yearly application modernization spend

Yearly application replatforming spend

The modernization choice

Manual rewriting:

Costs $2M+
Takes 6 - 16 months
Introduces bugs

The modernization choice

```
// (46)  77  CPT-IN        PIC S9(7) COMP-3   VALUE ZERO.
Var cpt_In =
    declare.level(77).picS9(7).comp3().valueZero().var();

// (47)  77  CPT-OUT       PIC S9(7) COMP-3   VALUE ZERO.
Var cpt_Out =
    declare.level(77).picS9(7).comp3().valueZero().var();

// (49)  77  FIN-TRAIT     PIC X          VALUE SPACE.
Var fin_Trait =
    declare.level(77).picX(1).valueSpaces().var();

// (51)  01  SYS-TIME      PIC 9(8)       VALUE ZEROS.
Var sys_Time = declare.level(1).pic9(8).valueZero().var();

// (52)  01  FILLER REDEFINES SYS-TIME.
Var filler$1 = declare.level(1).redefines(sys_Time).filler();

// (53)      03 SYS-TIME1  PIC 9(7).
Var sys_Time1 = declare.level(3).pic9(7).var();

// (54)      03 SYS-TIME2  PIC 9.
Var sys_Time2 = declare.level(3).pic9(1).var();
```

Automatic transpilation:

Fast
Cheap
Correct

But unmaintainable

The modernization choice

LLM-only rewrites:

Fast
Cheap
Maintainable

But introduces bugs and hallucinations

Docstring

```
# similarity_filter.py
Hydrator for `Time` and `LocalTime` values.
:param nanoseconds:
:param tz:
:return: Time
```

Ground-truth

```
def hydrate_time(nanoseconds, tz=None):
    from pytz import FixedOffset
    seconds, nanoseconds = map(int, divmod(nanoseconds, 1000000000))
    minutes, seconds = map(int, divmod(seconds, 60))
    hours, minutes = map(int, divmod(minutes, 60))
    t = Time(hours, minutes, seconds, nanoseconds)
    if tz is None:
        return t
    tz_offset_minutes, tz_offset_seconds = divmod(tz, 60)
    zone = FixedOffset(tz_offset_minutes)
    return zone.localize(t)
```

Handle `LocalTime` in functional requirement ✓

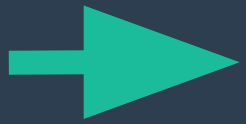
LLM Generation

```
def hydrate_time(nanoseconds, tz=None):
  from .time import Time
  return Time.from_nanoseconds(nanoseconds, tz)
```

Overlook `LocalTime` ✗

Our solution: Automated refactoring that is provably valid

```
// (46)  77  CPT-IN        PIC S9(7) COMP-3   VALUE ZERO.
Var cpt_In =
    declare.level(77).picS9(7).comp3().valueZero().var();

// (47)  77  CPT-OUT       PIC S9(7) COMP-3   VALUE ZERO.
Var cpt_Out =
    declare.level(77).picS9(7).comp3().valueZero().var();

// (49)  77  FIN-TRAIT     PIC X          VALUE SPACE.
Var fin_Trait =
    declare.level(77).picX(1).valueSpaces().var();

// (51)  01  SYS-TIME      PIC 9(8)       VALUE ZEROS.
Var sys_Time =
    declare.level(1).pic9(8).valueZero().var();

// (52)  01  FILLER REDEFINES SYS-TIME.
Var filler$1 =
    declare.level(1).redefines(sys_Time).filler();

// (53)      03 SYS-TIME1  PIC 9(7).
Var sys_Time1 = declare.level(3).pic9(7).var();

// (54)      03 SYS-TIME2  PIC 9.
Var sys_Time2 = declare.level(3).pic9(1).var();
```



```
long counter = 0;

long result = 0;

String endMarker = " ";

unsigned long sysTime = 0;

unsigned long sysTime1() {
    return sys_Time / 10;
}

unsigned long sysTime2() {
    return sys_Time % 10;
}
```

Our solution

Step 1

LLM generates transforms

LLMs produce a large library of refactoring transforms, which can later be applied to any codebase we encounter.

Step 2

Formally verify the transform

We formally verify the transform offline to ensure that it preserves the semantics of the original code.

Step 3

Receive transpiled code

We receive code that has been transpiled into Java from another language using off-the-shelf transpilers.

Step 4

LLM sequences the transforms

IBM Granite takes our list of formally verified transforms and sequences them in the right order to produce high-quality code, that's provably identical to the original.

Long-term vision: Universal paradigm translator

There’s nothing unique about Java or legacy languages like COBOL.

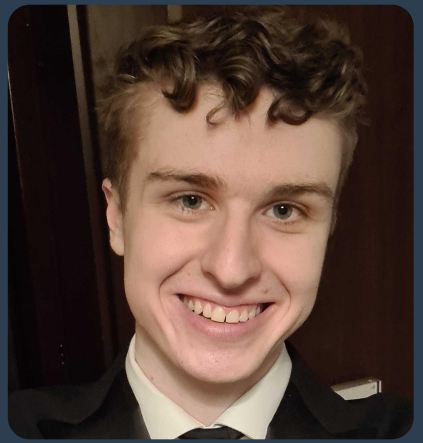
The underlying technology could allow us to produce highly-maintainable, provably valid translations from any language to any other language— automatically.

Kerry Vaughan-Rowe

YC W17
PhD in philosophy

Christopher Little-Savage

Cambridge CS MEng
10yrs startup experience



Joe O'Connor

Cambridge CS MEng
Prev. Bloop (YC S21)

Validly

Modern-day code modernization

Revenue Opportunities

Company-first

Target large companies that need to modernize.

Use partnerships to handle other aspects of modernization.

Consultancy-first

Partner directly with consultancies that handle large-scale modernization.

We are responsible for only the code base.