All code is buggy. It stands to reason, therefore, that the more code you have to write the buggier your apps will be.

Writing more code also takes more time, leaving less time for other things like optimisation, nice-to-have features, or being outdoors instead of hunched over a laptop.

In fact it's widely acknowledged that project development time and bug count grow *quadratically*, not linearly, with the size of a codebase. That tracks with our intuitions: a ten-line pull request will get a level of scrutiny rarely applied to a 100-line one. And once a given module becomes too big to fit on a single screen, the cognitive effort required to understand it increases significantly. We compensate by refactoring and adding comments — activities that almost always result in *more* code. It's a vicious cycle.

Yet while we obsess — rightly! — over performance numbers, bundle size and anything else we can measure, we rarely pay attention to the amount of code we're writing.

## Readability is important

I'm certainly not claiming that we should use clever tricks to scrunch our code into the most compact form possible at the expense of readability. Nor am I claiming that reducing *lines* of code is necessarily a worthwhile goal, since it encourages turning readable code like this...

```
for (let i = 0; i <= 100; i += 1) {
    if (i % 2 === 0) {
        console.log(`${i} is even`);
    }
}
```

...into something much harder to parse:

```
for (let i = 0; i <= 100; i += 1) if (i % 2 === 0) console.log(`${i} is even`);
```

Instead, I'm claiming that we should favour languages and patterns that allow us to naturally write less code.

## Yes, I'm talking about Svelte

Reducing the amount of code you have to write is an explicit goal of Svelte. To illustrate, let's look at a very simple component implemented in React, Vue and Svelte. First, the Svelte version:

404

The requested path could not be found

How would we build this in React? It would probably look something like this:

```
import React, { useState } from 'react';

export default () => {
    const [a, setA] = useState(1);
    const [b, setB] = useState(2);

    function handleChangeA(event) {
        setA(+event.target.value);
    }

    function handleChangeB(event) {
        setB(+event.target.value);
    }

    return (
        <div>
            <input type="number" value={a} onChange={handleChangeA}/>
            <input type="number" value={b} onChange={handleChangeB}/>

            <p>{a} + {b} = {a + b}</p>
        </div>
    );
};
```

Here's an equivalent component in Vue:

```
<template>
    <div>
        <input type="number" v-model.number="a">
        <input type="number" v-model.number="b">

        <p>{{a}} + {{b}} = {{a + b}}</p>
    </div>
</template>

<script>
    export default {
        data: function() {
            return {
                a: 1,
                b: 2
            };
        }
    };
</script>
```

I'm counting by copying the source code to the clipboard and running `pbpaste | wc -c` in my terminal

In other words, it takes 442 characters in React, and 263 characters in Vue, to achieve something that takes 145 characters in Svelte. The React version is literally three times larger!

It's unusual for the difference to be *quite* so obvious — in my experience, a React component is typically around 40% larger than its Svelte equivalent. Let's look at the features of Svelte's design that enable you to express ideas more concisely:

### Top-level elements

In Svelte, a component can have as many top-level elements as you like. In React and Vue, a component must have a single top-level element — in React's case, trying to return two top-level elements from a component function would result in syntactically invalid code. (You can use a fragment — `<>` — instead of a `<div>`, but it's the same basic idea, and still results in an extra level of indentation).

In Vue, your markup must be wrapped in a `<template>` element, which I'd argue is redundant.

### Bindings

In React, we have to respond to input events ourselves:

```
function handleChangeA(event) {
    setA(+event.target.value);
}
```

This isn't just boring plumbing that takes up extra space on the screen, it's also extra surface area for bugs. Conceptually, the value of the input is bound to the value of `a` and vice versa, but that relationship isn't cleanly expressed — instead we have two tightly-coupled but physically separate chunks of code (the event handler and the `value={a}` prop). Not only that, but we have to remember to coerce the string value with the `+` operator, otherwise `2 + 2` will equal `22` instead of `4`.

Like Svelte, Vue does have a way of expressing the binding — the `v-model` attribute, though again we have to be careful to use `v-model.number` even though it's a numeric input.

### State

In Svelte, you update local component state with an assignment operator:

```
let count = 0;

function increment() {
    count += 1;
}
```

In React, we use the `useState` hook:

```
const [count, setCount] = useState(0);

function increment() {
    setCount(count + 1);
}
```

This is much *noisier* — it expresses the exact same concept but with over 60% more characters. As you're reading the code, you have to do that much more work to understand the author's intent.

In Vue, meanwhile, we have a default export with a `data` function that returns an object literal with properties corresponding to our local state. Things like helper functions and child components can't simply be imported and used in the template, but must instead be 'registered' by attaching them to the correct part of the default export.

## Death to boilerplate

These are just some of the ways that Svelte helps you build user interfaces with a minimum of fuss. There are plenty of others — for example, reactive declarations essentially do the work of React's `useMemo`, `useCallback` and `useEffect` without the boilerplate (or indeed the garbage collection overhead of creating inline functions and arrays on each state change).

How? By choosing a different set of constraints. Because Svelte is a compiler, we're not bound to the peculiarities of JavaScript: we can *design* a component authoring experience, rather than having to fit it around the semantics of the language. Paradoxically, this results in *more* idiomatic code — for example using variables naturally rather than via proxies or hooks — while delivering significantly more performant apps.