Keeping a System Running Forever

how I keep systems running

“forever”

everything is terrible

the network is terrible

the libraries are buggy

communication between humans is hard

the specifications will be wrong

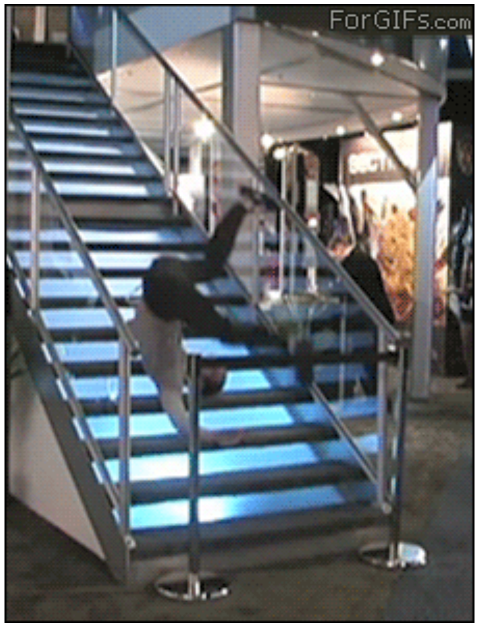
I make mistakes all the time

prepare for the worst case

“I'm selfish, impatient and a little insecure. I make mistakes, I am out of control and at times hard to handle. But if you can't handle me at my worst, then you sure as hell don't deserve me at my best.”

- ~~Marilyn Monroe~~

- My Software



start (and restart) safely

ground rules: ugh, state!

state is the most important thing

state is also where pain lives

get rid of bad state

know what state you can go back to

it's all about the guarantees

```
init(Args) ->
    State = init_state(Args),
    {ok, NewState} = connect(State),
    {ok, NewState}.

[...]

handle_info(reconnect, State) ->
    case connect(State) of
        {ok, NewState} -> {noreply, NewState};
        _ -> self() ! reconnect, {noreply, S}
    end;
```

it’s all about the guarantees

```
init(Args) ->
    %% don’t guarantee connections
    self() ! reconnect,
    {ok, init_state(Args)}.

[...]

handle_info(reconnect, State) ->
    case connect(State) of
        {ok, NewState} -> {noreply, NewState};
        _ -> self() ! reconnect, {noreply, S}
    end;
```

it’s all about the guarantees

you can’t guarantee what you don’t control

you **steal** control on these issues

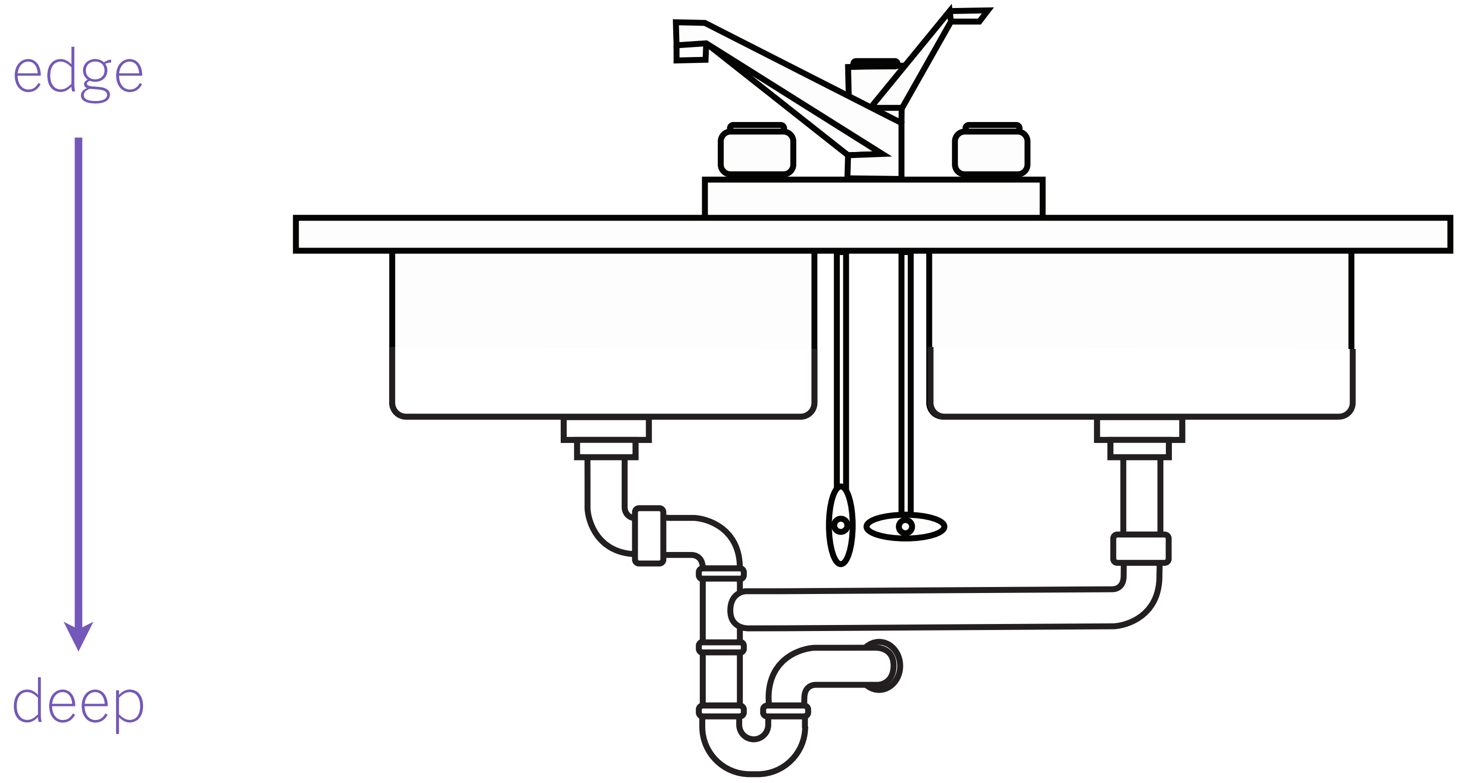
BUT

it’s useless to boot fast if you boot wrong

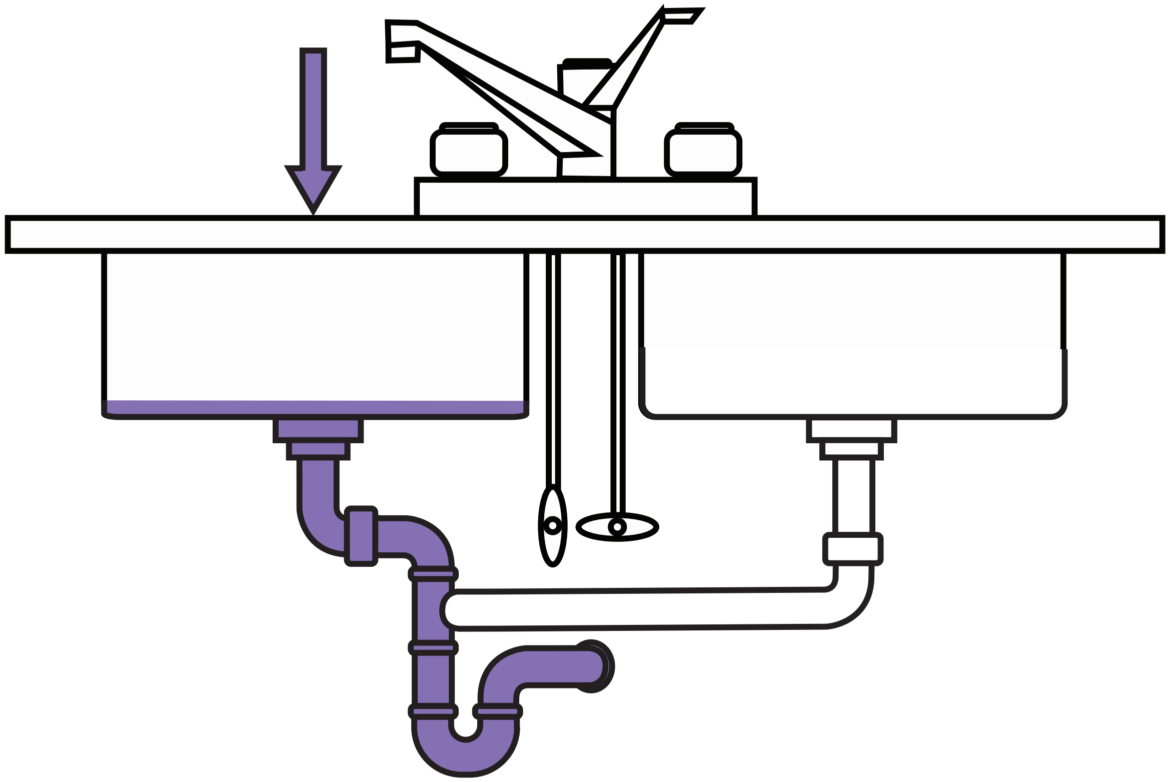
it’s useless to boot **at all** if you boot wrong

Plan for Overload

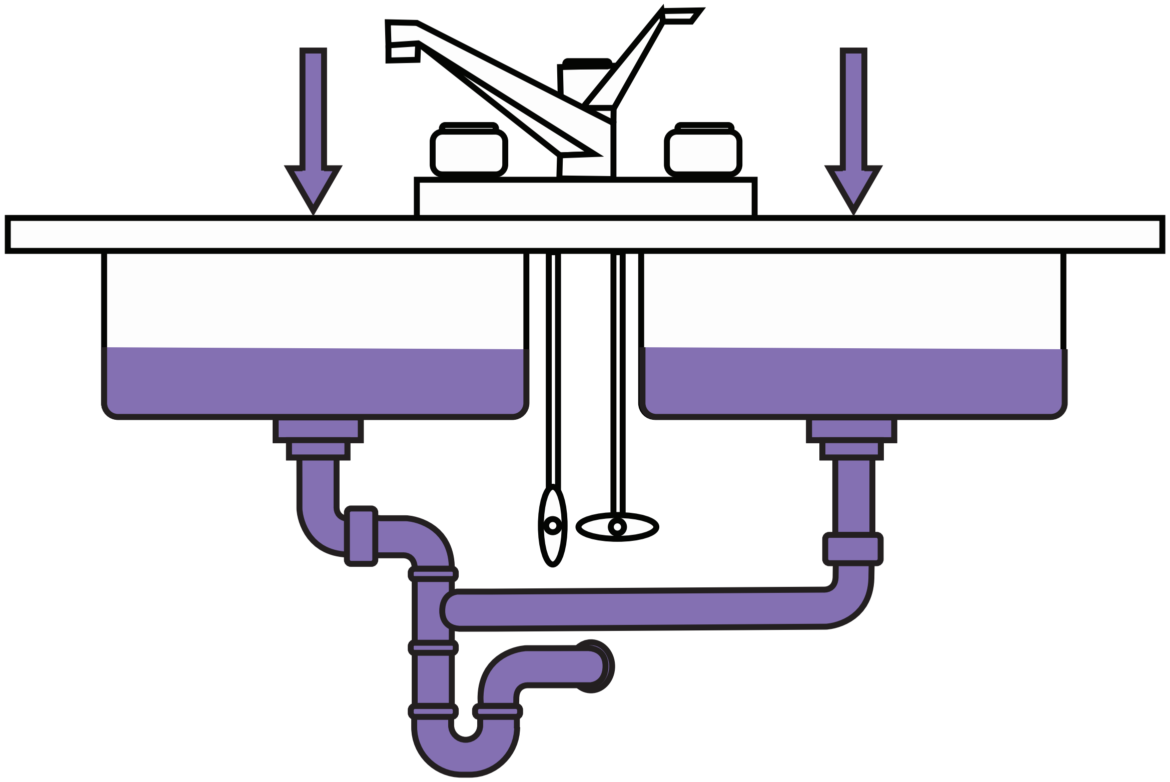
your system is a bathroom sink



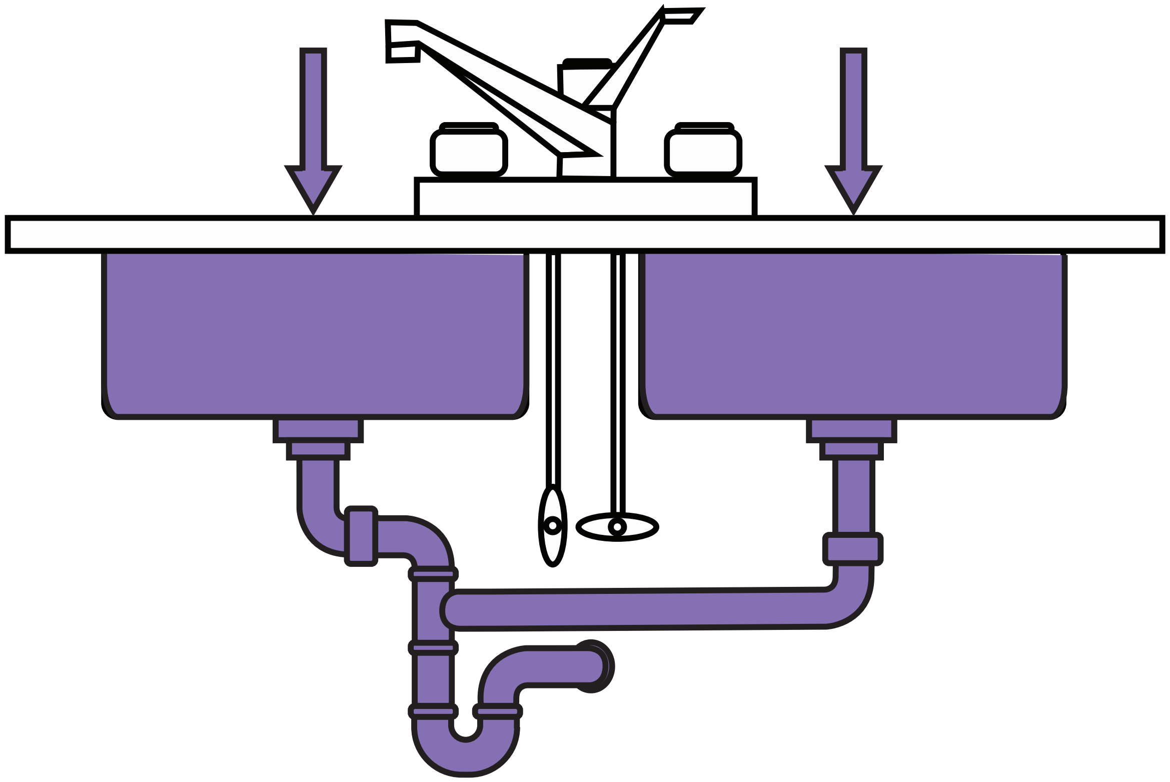
normal operations



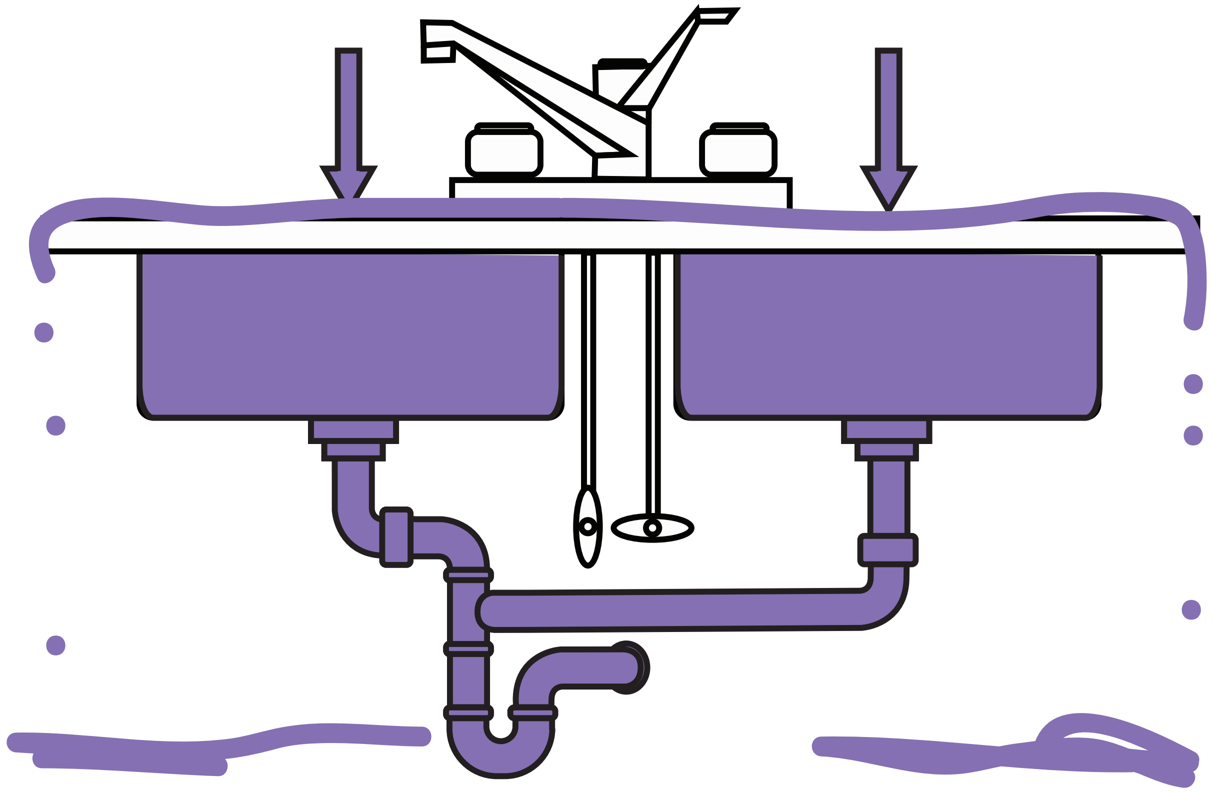
temporary overload



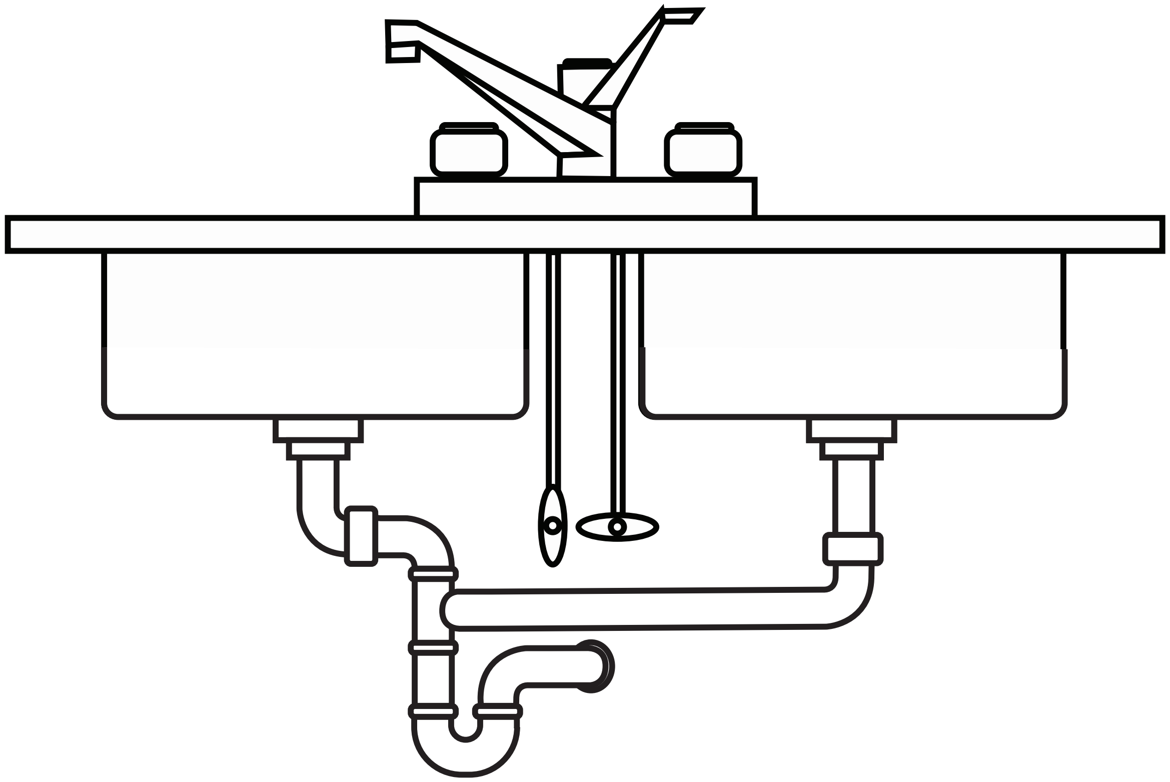
prolonged overload



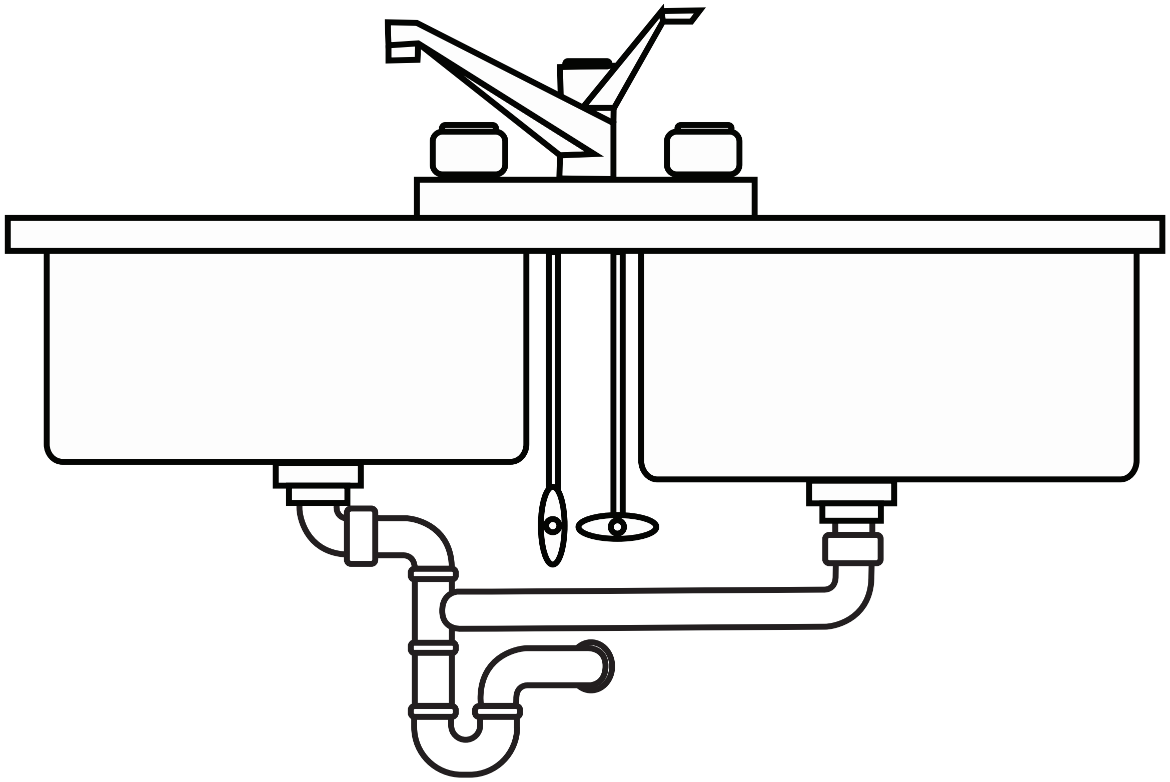
crash dump!



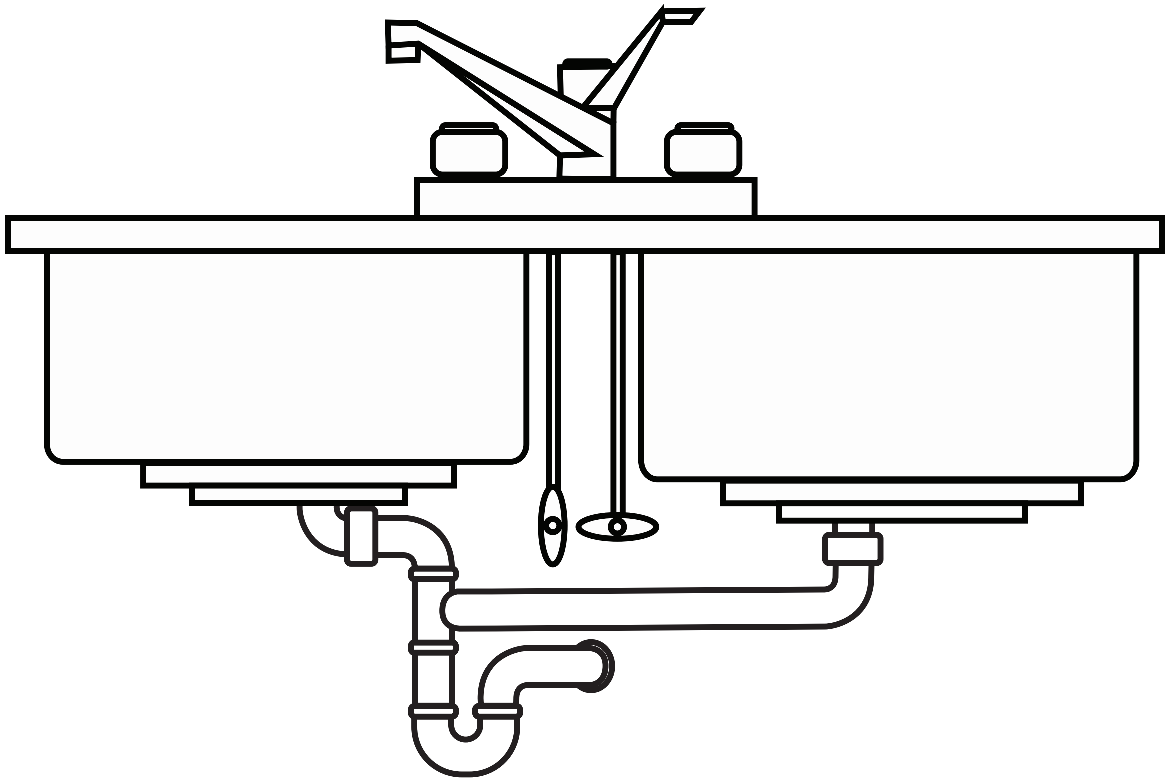
if we make it bigger, it’s gonna handle more flow optimize away!



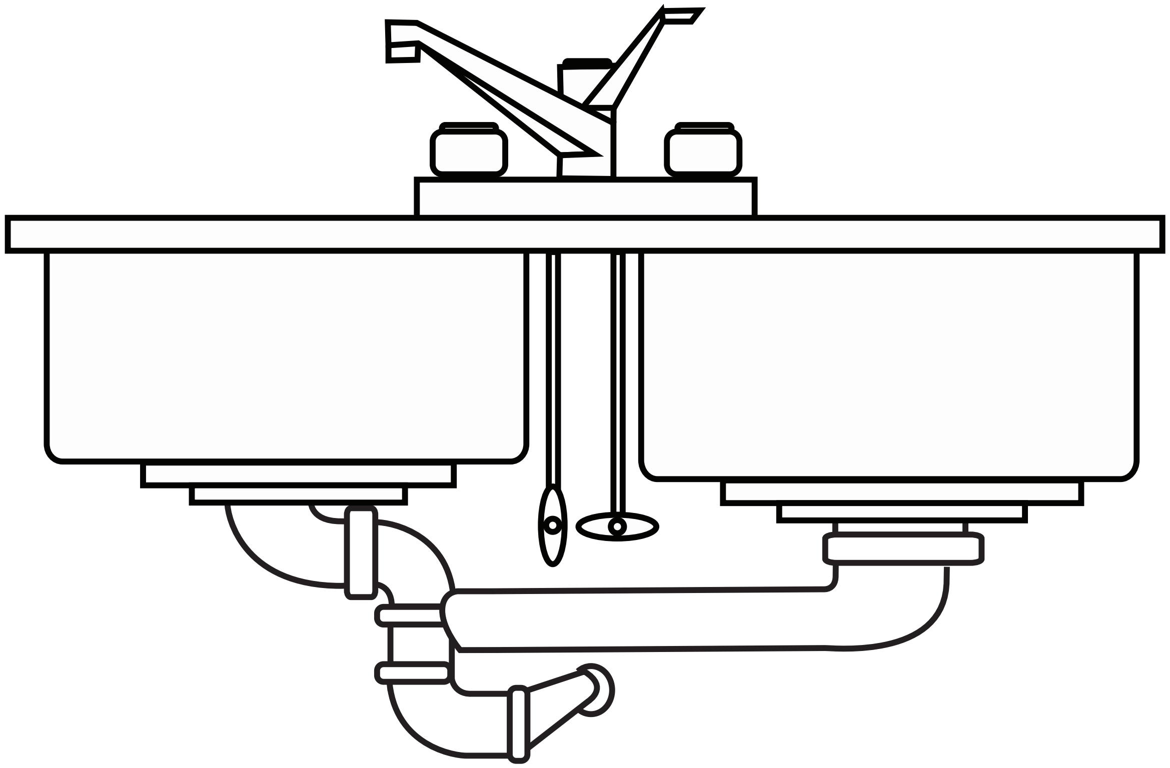
bigger sinks!



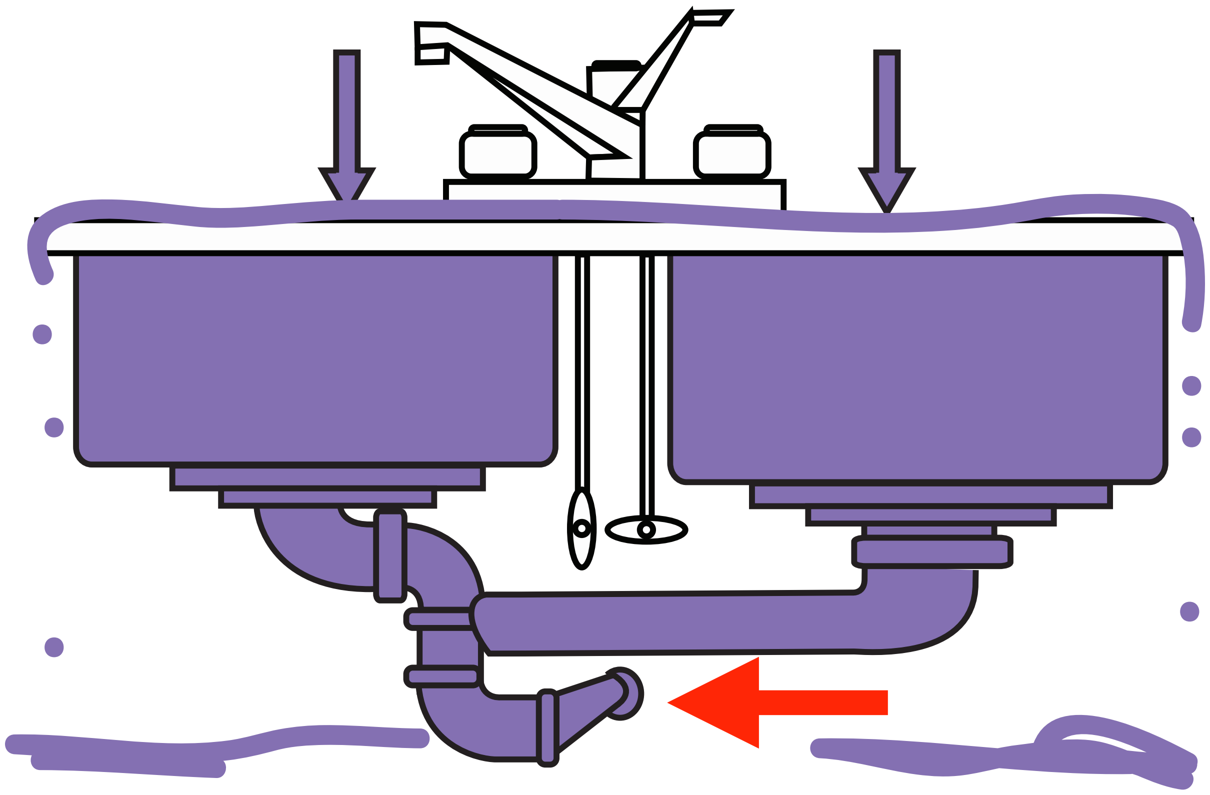
bigger drains!



bigger pipes!



bottlenecks you don't control



paid to solve the wrong problem

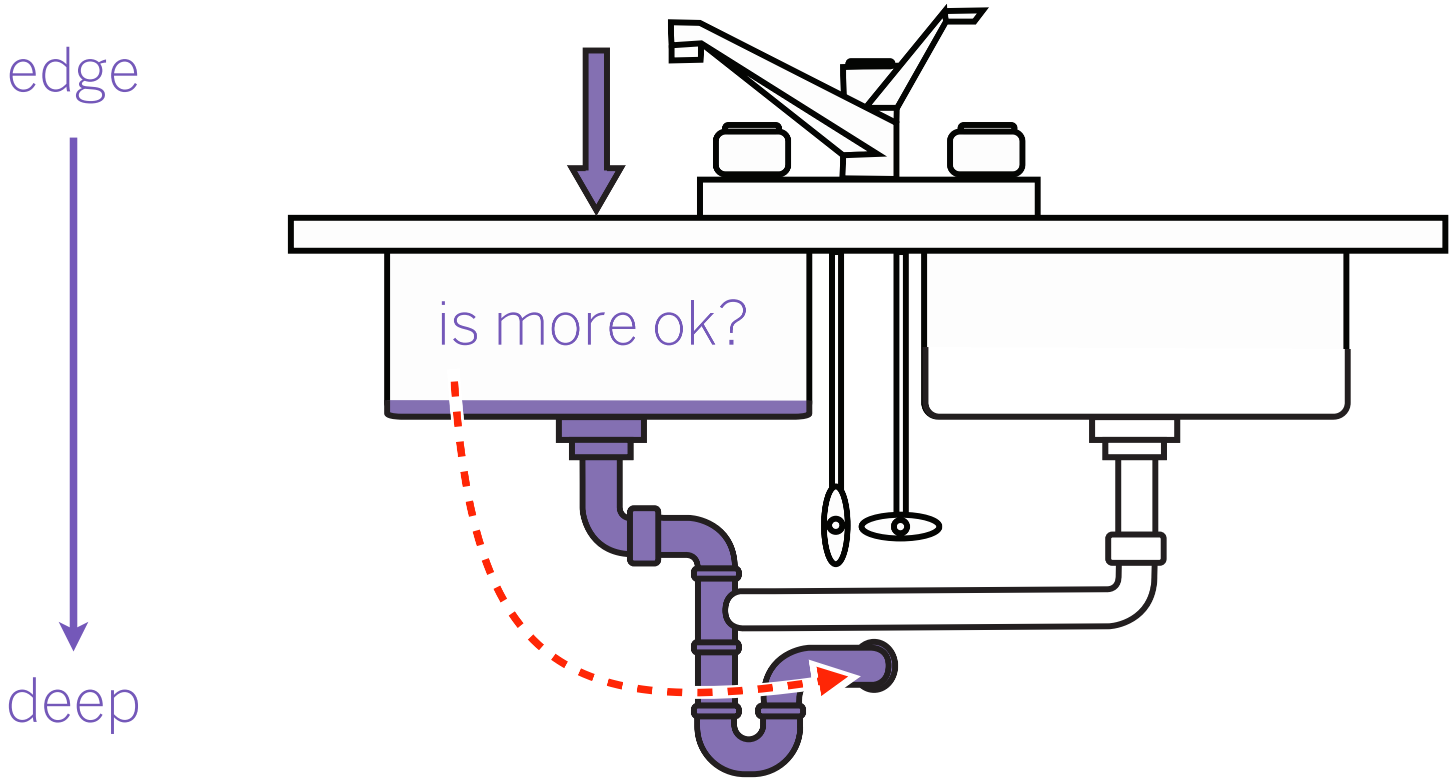
pick what has to give

block on input
(back-pressure)

drop data on the floor
(shed load)

it’s a business decision

ask for permission



random drop

when some loss is acceptable (sample size!)

can be made adaptive

works even better producer-side

```
case drop:random(0.95) of
    true -> send();
    false -> drop()
end
```

```
random(Rate) ->
    maybe_seed(),
    random:uniform() =< Rate.
```

queue buffers

more control than random drop

can drop from either end of the queue if full

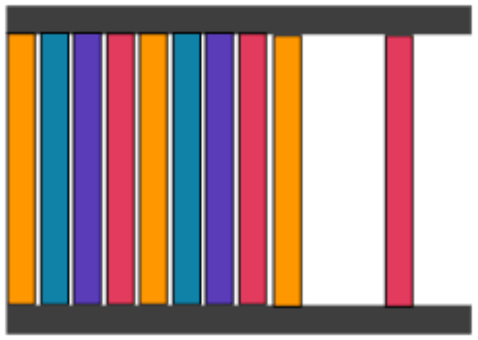
useful if you need messages in order

stack buffers

better for low latency

no requirement for ordering

discard oldest data, or all data too old



overload handling

use processes or ETS tables to ask permission

os_mon, SASL

https://github.com/jlouis/safetyvalve

https://github.com/uwiger/jobs

https://github.com/klarna/circuit_breaker

https://github.com/ferd/pobox

how do you tell users?

Respect End-to-End principles

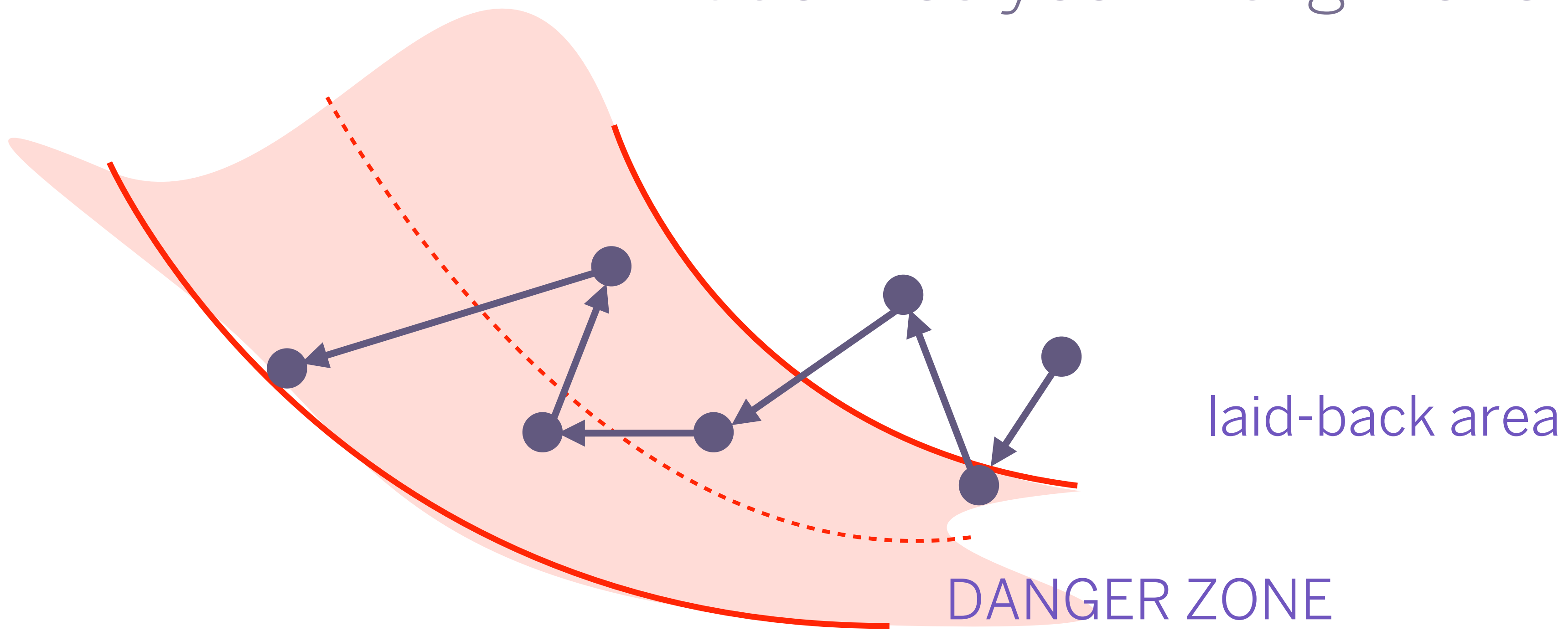
Make idempotent APIs

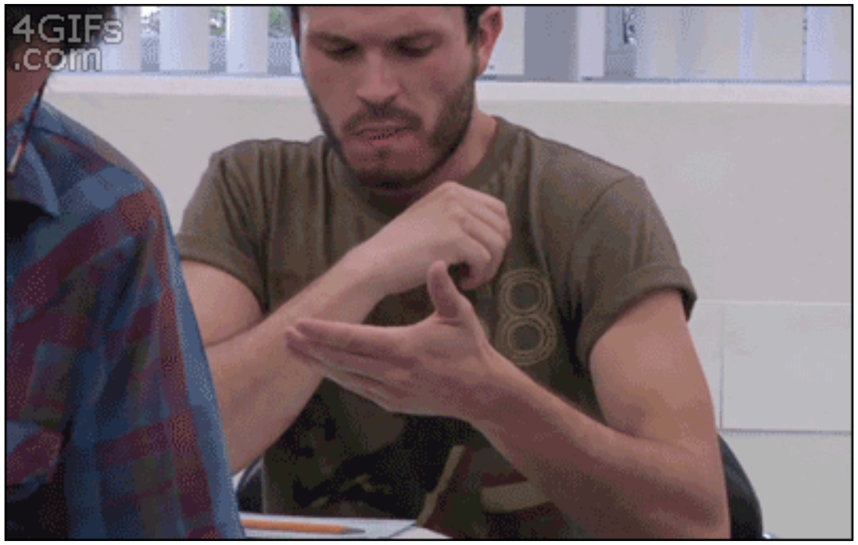
Tell about losses

Put usage limits, however high

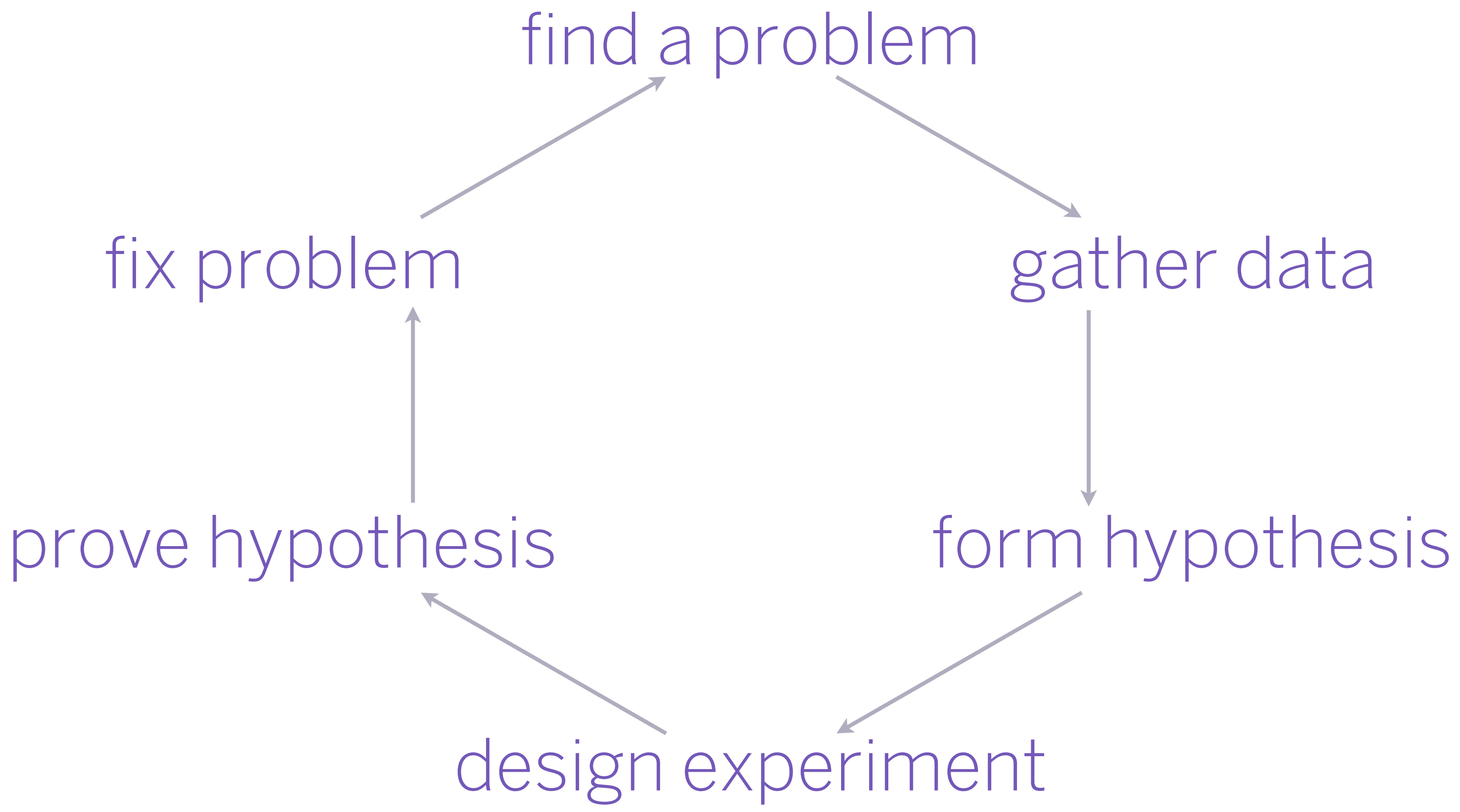
Overload must be planned for

it defines your margin of error





be ready to get your hands dirty


find a problem
gather data
form hypothesis
design experiment
prove hypothesis
fix problem

introspect everything

traces, processes, GC, memory, the network, other nodes

ERLANG-IN-ANGER.COM

http://jobs.heroku.com/