Richard Carlsson

# The Art of the Live Upgrade

Lessons from 10 years of evolving a live system

# Klarna

- Small startup in Sweden in 2005 - I joined in 2008
- Erlang-based system
- Online payments
- Intermediate between customer and merchant, making it simpler to buy
- Settle payment later (using card, bank transfer, invoice, …)
- Now in many countries, including Germany, UK, US
- Millions of customers
- Not only Erlang anymore; many different systems
- We’re now a bank - traceability and isolation required

# Our system

# The Kred system

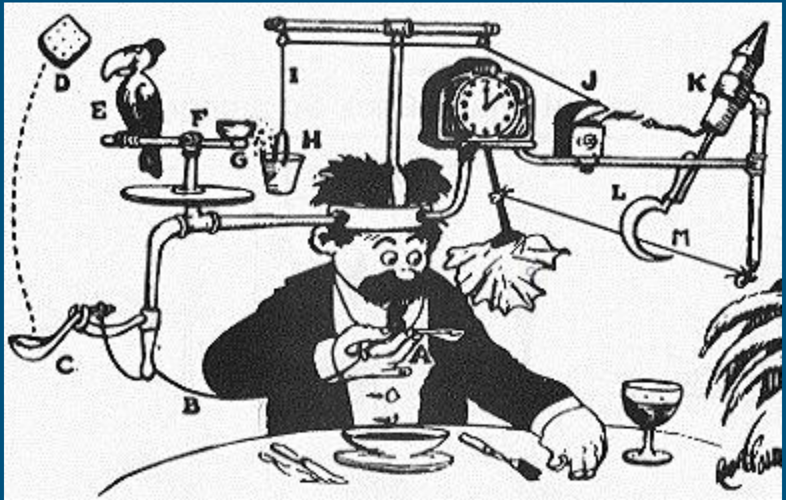
- Named after the original company name, *Kreditor*.
- Monolithic - originally handled everything from receiving purchases and doing bookkeeping to serving the web interface and printing invoice pdf:s.
- These days, several parts have moved out to separate services.
- Went live 2005 - 14 years of 24/7 availability.
- No scheduled downtime, ever.
- Never been down for more than about 2 hours consecutive.
- Not perfect - about 99.98% availability per year, historically. (1-2 h/yr)
- Still a lot to do - we want those 5 nines...

# Kred system vital stats

- Approx. 1 million lines of Erlang code (not counting tests)
- About 250 separate applications (~20% external open source apps)
- Close to 4000 modules (not counting tests)
- Around 60 developers today, spread over half a dozen teams
- Over 200 committers over the years
- Git history preserved all the way back to late 2004 (CVS -> SVN -> Git)
- Around 140 thousand commits

# CI Pipeline

- Git
- Bitbucket
- Jira
- Jenkins
- Docker
- Ansible
- AWS
- RPM
- Artifactory



# Pipeline and Branching model

- Git, Bitbucket, Jira, Jenkins, Docker, AWS
- Master is current live version
- Stable will be the next Master (fast-forward), safe to base work on
- Devs branch off Master or Stable
- Every branch must have a Jira ticket (we’re a bank...)
- Builds and runs test suites automatically when pushed to Bitbucket
- Code reviews: at least 2 approvals per branch (we’re a bank…)
    - Path-based: any changes to a team’s code must be approved by someone in that team
- Ready branches get merged and integration tested, Stable moves forward

# Nightly tests and releases

- Every evening, the current Stable is tagged as a Release Candidate
- We throw all our system tests at it (4-5 hours…)
- Live-like test environment
  - Multiple nodes, full-size test db, simulated traffic, …
  - Performance graphs, see if something has changed radically
- If everything looks good in the morning, we take the release live
- Typically 2-12 tickets pushed out per day
- Good turnaround - most branches only live for a few days

# Problems in the pipeline

- Sometimes the pipeline gets clogged and nothing is released for a few days
- A sneaky error can make it all the way into the integration or system testing
    - That’s what the testing is for - to find them before they go live
    - New ticket and PR to fix the Stable branch again (might take 1-2 hours)
- Main cause of disruptions is hiccups in AWS, Jenkins, Artifactory, Bitbucket, … - a single failure there can lose a whole night’s build/test.
- Sometimes time for a rerun and release in the afternoon
    - Hope to be able to speed up things even more to release multiple times per day.

# Quick fixes and configuration changes

- Fix urgent problems on live systems (bugs, environmental changes)
  - Bypass the main release pipeline
  - Still needs ticket, review, stakeholder approval
  - Needs to pass build and test suites
  - Separate Change Request to Live Operations (we’re a bank…)
  - Deploy code as for any other upgrade
- Configuration changes on live systems
  - Need ticket and Change Request to Live Operations (we’re a bank…)
- With good tools it only takes minutes to handle the extra administration
  - Not a lot of pain
  - Traceability is nice for developers too, a couple of years later!

# Keeping developers happy/productive

- Keeping local compile times down (heavily parallel build)
- Reducing pipeline turnaround time (parallelized test suites)
- Tool integration, reduce manual clicking in web interfaces
- Code ownership - Erlang Apps owned by teams
  - Reduce likelihood of merge conflicts
  - Easy to determine who reviews what
- Applications make it easier to add and remove whole code units
  - Satisfying to delete a whole application
- Find silly errors at local build time, not at the end of a full test suite run
  - Xref checks, etc.

# Reshaping our system

# A different time when we started

- OTP 10
- No Common Test, no Rebar, no Dialyzer
- No cloud computing
- Yaws was the obvious/only choice for a web server
- No good database bindings, so Mnesia was the obvious choice
- Nobody really knew how OTP releases worked
- CVS/Subversion (switched to Git in mid-2010, helped a lot with merges)
- Cruise Control for automated build/test

# We made every mistake in the book

- ...and some of those books were not written back then.
- Erlang/OTP and the Beam have held up incredibly well!
- Handled an enormous growth that nobody had expected or dreamt of
- Erlang has allowed us to restructure bit by bit without stopping

# From few applications to many

- Original system had ~10 apps (of which 2 were ‘misc’ and ‘util’)
- By late 2011 we had about 30 apps, and about 100 developers
- Merge conflicts, code ownership, dependencies, code structure, ...
- “Cambrian explosion”: Split into 100+ apps (without stopping anything)
- Made it easy to add/remove apps as units of functionality - now ~250

# Splitting up bad supervision trees

- Some of the original apps were doing too many things
- Several unrelated processes under the same supervision tree
- Moved out to new apps with their own supervision
  - Some processes could simply be stopped and restarted under new supervision
  - Some critical processes were moved to new supervisors without stopping
- Made it easier to think about responsibilities of a particular app

# Reducing coupling

- Splitting up header files (small is good) - avoid useless recompiling
- Parallelize build as much as possible
- Applications split into layers; xref-like checking of app dependencies
  - Internal tool, might make open source on day
- Code which is only intended for testing should not be part of release
  - Ensure nobody relies on some “for test only” functionality on live systems
- More abstraction - restrict knowledge to specific modules; others use API
  - Data structures
  - Database tables
- Break out subsystems to external services (maybe not in Erlang)

# Live upgrades

# How to do a live upgrade

1. Deploy code to machines
2. Load it
3. Profit!

But why not just stop servers and restart them one at a time?

# Why do live upgrades?

- If you stop one node, you need at least 2 more to guarantee redundancy
  - You might not care if you have a bunch of nodes in the cloud
  - On a black-box telecom server or embedded system, it’s a matter of economics
  - Stopping and restarting can take a lot of time
  - You might not want a long window where nodes are running different code
- Live upgrades can have much less impact on the system

# Enable small changes without stopping

- Bug fixes or temporary workarounds
- Add missing logging or instrumentation when you need it
- Insert redirections of calls to switch to a new service
  - First running in shadow mode, to verify that it behaves as expected
  - Start directing percentage of traffic to new service
  - Increment until all load is on new service
  - Disable old service
  - Remove obsolete code

# We don’t use OTP releases...

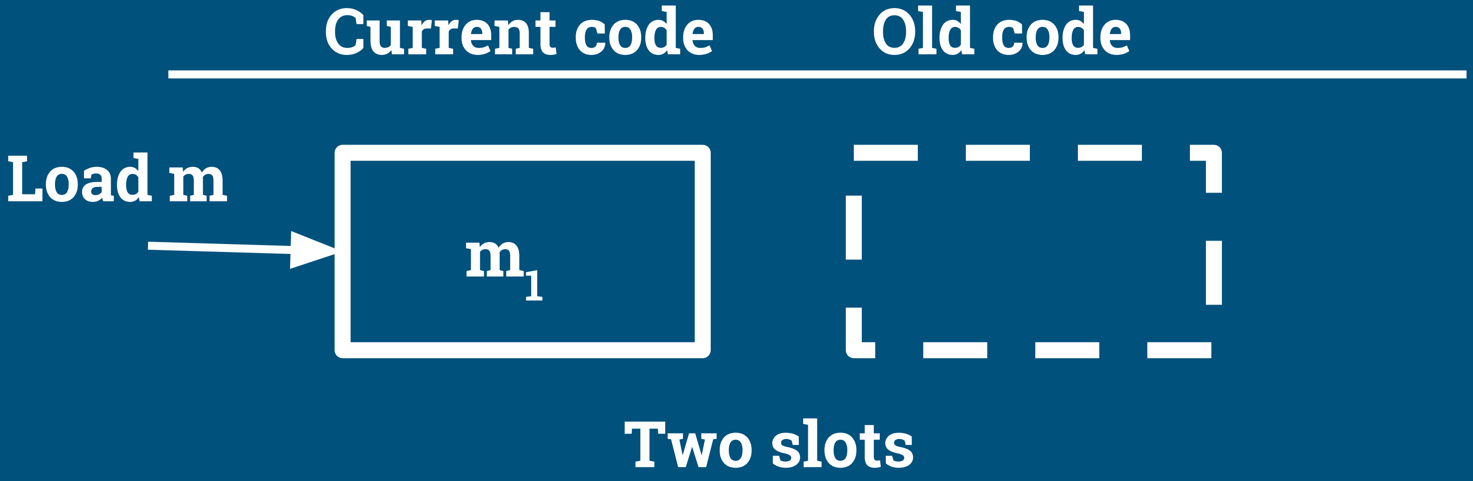
- Historical reasons
  - Releases were specialized knowledge even for us
  - Poor documentation and tools
- May switch to releases using Rebar3 in the future, but no hurry
- Releases use a static list of modules to be loaded, paths to add
- We detect dynamically which modules have been changed and which apps need to be added to the code path
  - Originally based on Beam file timestamps
  - Nowadays, using the MD5 check of `code:modified_modules()`
- “Upgrade actions” for things beyond simple code changes
- We don’t suspend gen_servers during upgrade

# Erlang code management

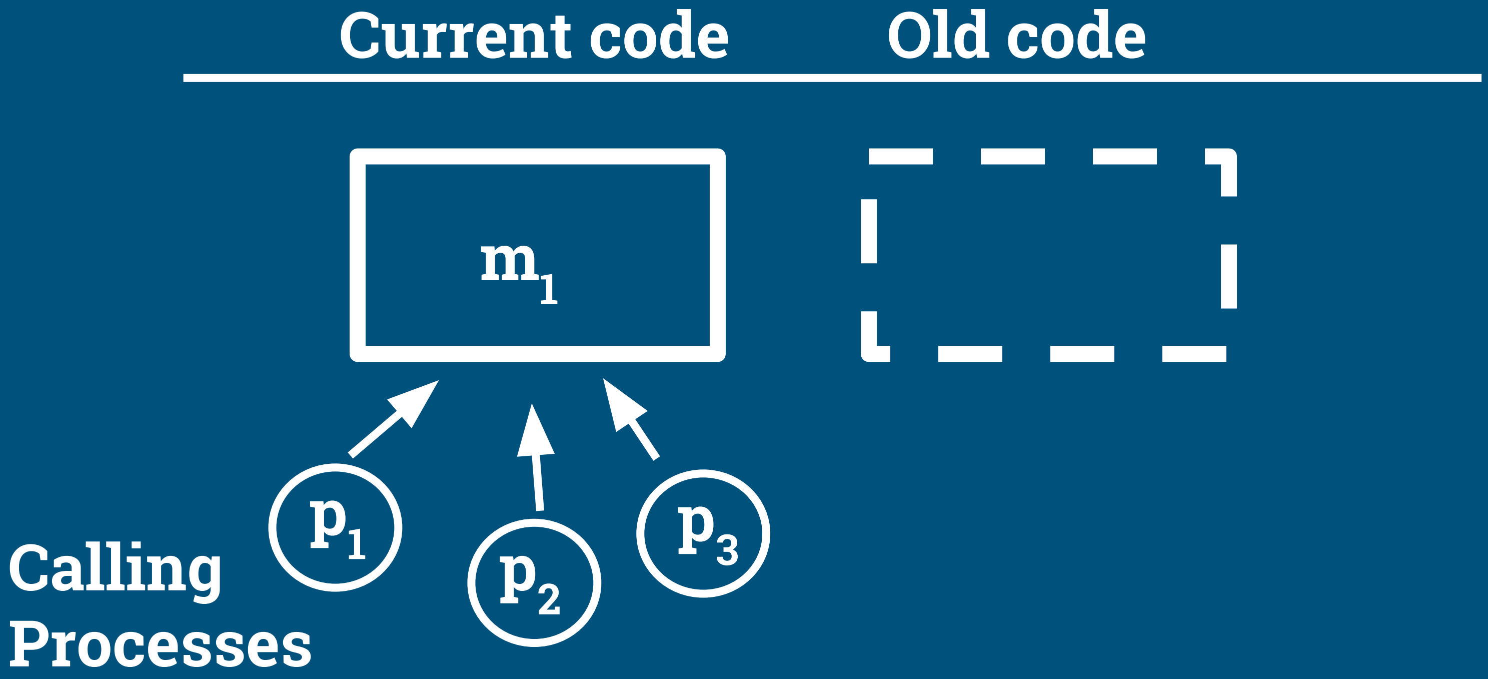
# Fundamental Design Decisions of Erlang

- Very long running systems (telecom switches)
- Upgrade code without stopping the system
- Modules as the unit of code delivery
- At most two concurrent versions
    - No memory leaks as you keep loading new versions
    - No older code running (no one still using older data formats or protocols)

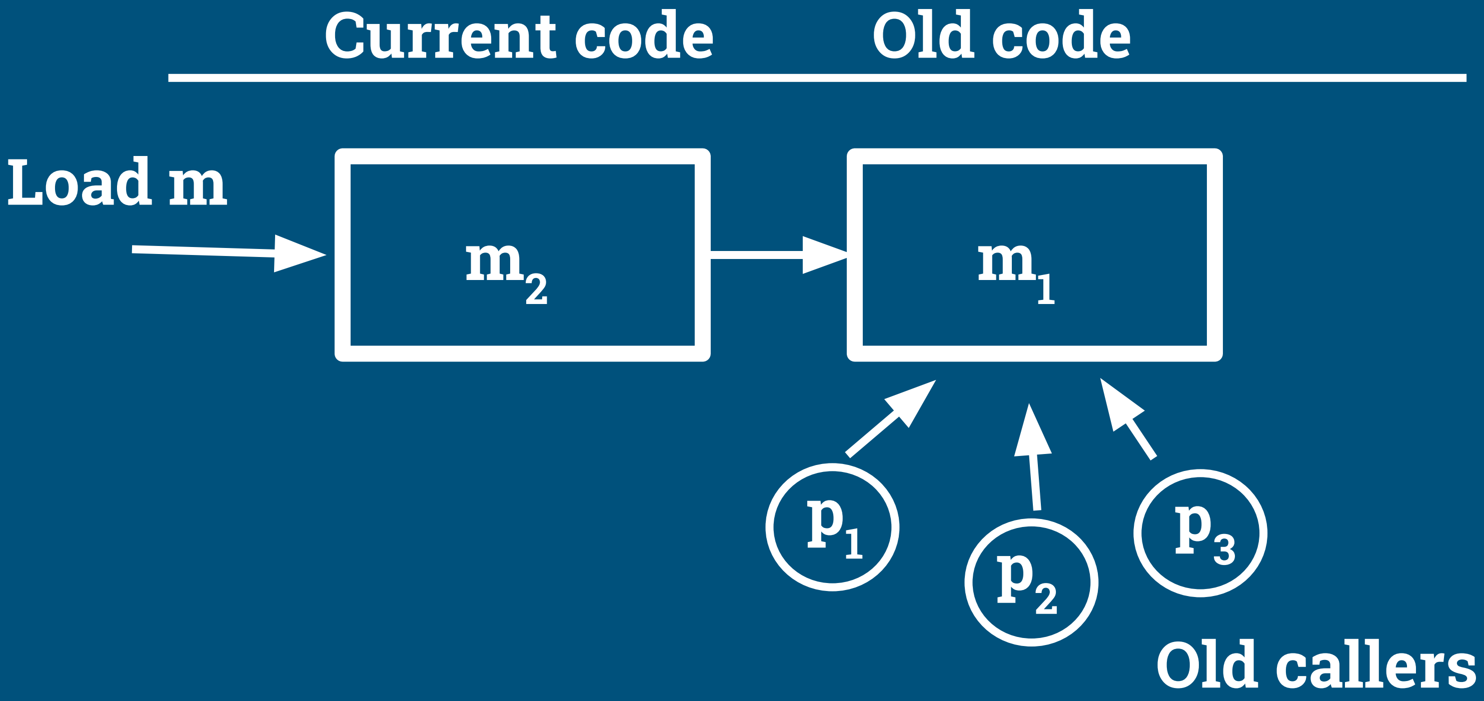
# Loading an Erlang module



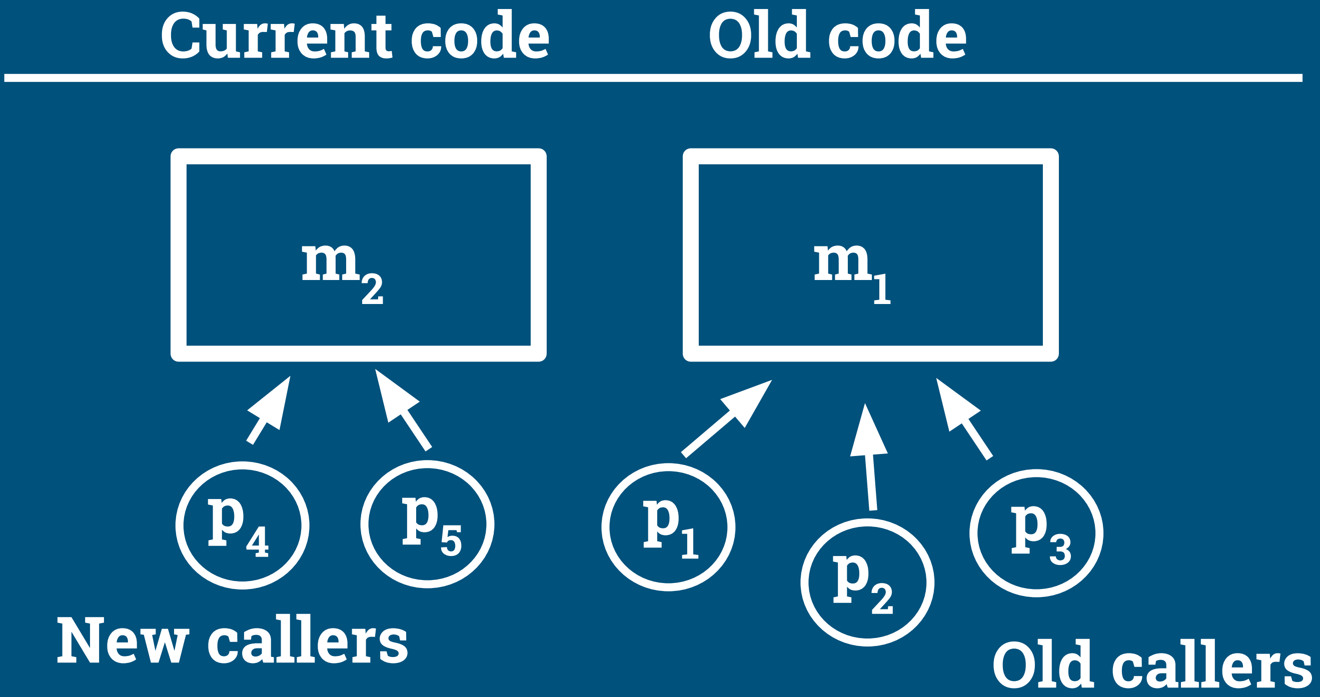
# Qualified calls m:f() go to the current version



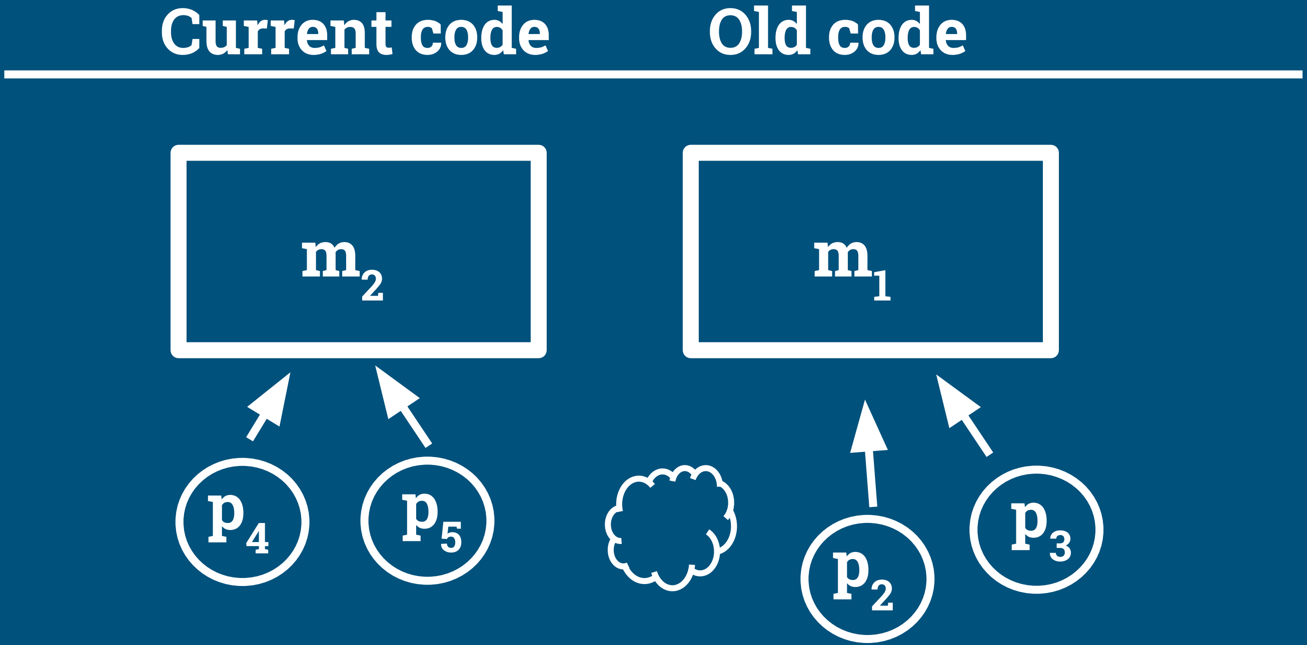
# Loading a new version (atomic)



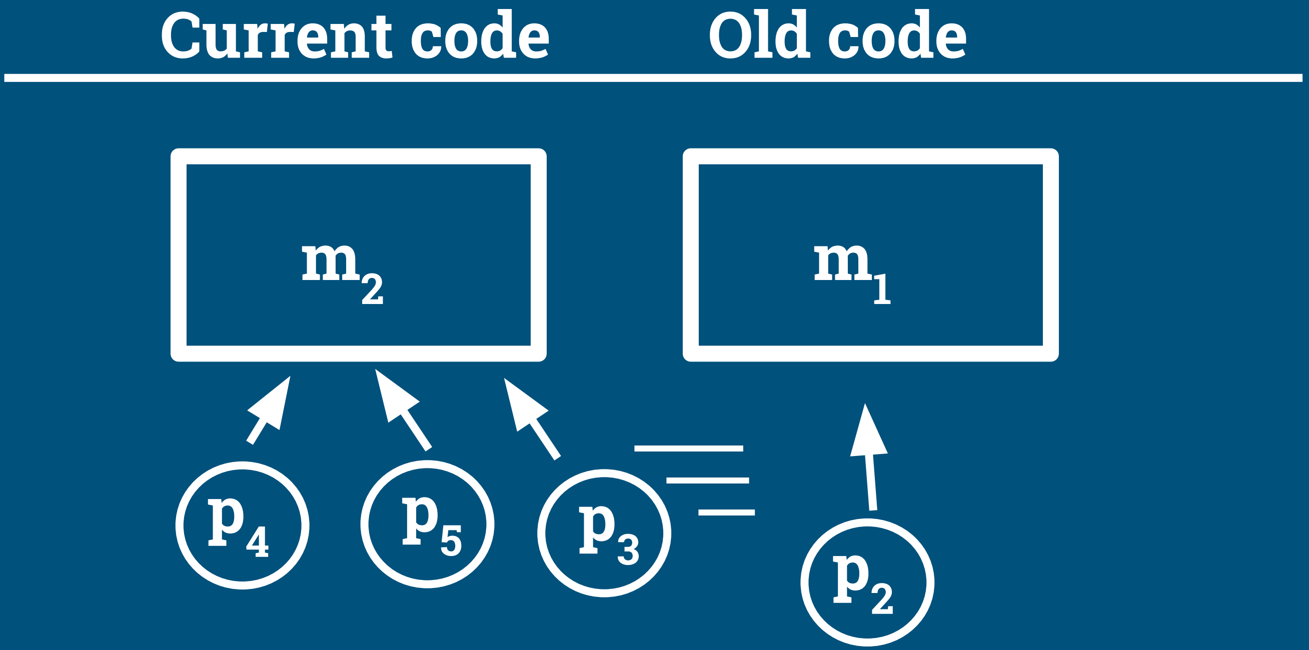
# New callers will use the new version



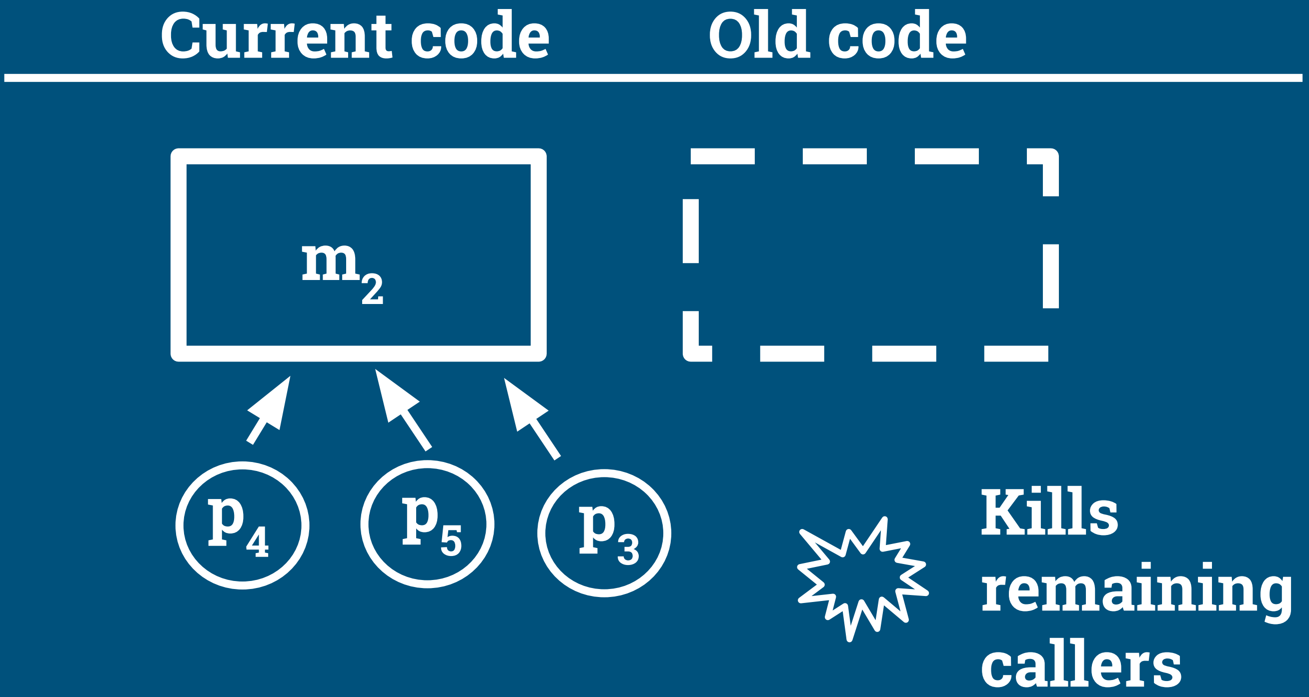
# Old callers terminate (finish or crash)...



# ...or migrate by a qualified tail call m:f()



# Old slot must be purged to load again



Current code

Old code

$m_2$

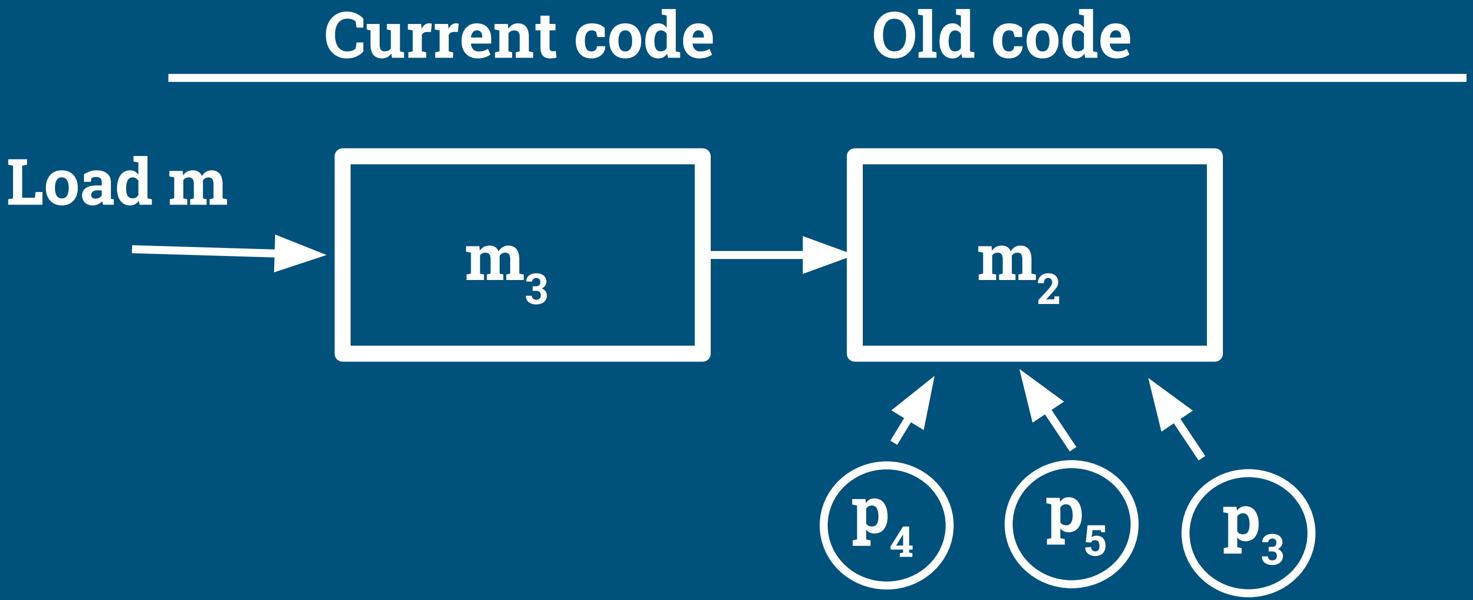
$p_4$

$p_5$

$p_3$

Kills remaining callers

# Load next version (and so on)



# Code pointers

- Current program point of a process
- Program points saved on the process' call stack
- Referencing a specific internal point in a specific version of the module
- To migrate a process to new code, you must:
  - Perform a tail call - do not leave any pointers to the old code on the stack
  - ...using a qualified ("remote") call m:f(...) - so you jump to the new code
- Any long-running process should regularly go via such a call
  - E.g., a server loop typically does this between handling requests
- Funs are not code pointers, handled differently

# The Code Server: `code:*(...)`

- Manages the code path
- Tracks additional info
  - Path from where the Beam code was loaded
- Sticky directories and modules
  - Prevents accidentally replacing system modules
- Other utility functions
  - Look up application directories, e.g. lib_dir(stdlib, ebin)
  - Check for module name clashes in your path
  - Check which modules are modified on disk

# Basic code operations

- `code:load_file(ModName)`
  - Loads from code path, atomically moves any current code to Old slot
  - Old slot must be purged first; only the shell function `l(ModName)` will purge for you
- `code:load_binary(ModName, Filename, Binary)`
  - Load a Beam binary object, Filename is just metadata - use empty string for generated code
- `code:purge(ModName)`
  - Purges old code slot. Kills any processes still referencing the old code
- `code:soft_purge(ModName)`
  - Gives up and returns false if some process still references the old code
- `code:delete(ModName)`
  - Moves current code to the old slot (must be purged first), no new calls can be made

# Fully purging a module

- To remove a module and ensure no process is still referencing that code:

    `purge(ModName)` - drop any existing old version

    `delete(ModName)` - move current to old slot, clearing current

    `purge(ModName)` - clear old slot

# Detecting modified modules

- `code:modified_modules()`
  - List all loaded modules that have changed on disk
- `code:module_status(ModName)`
  - `-> not_loaded | loaded |modified| removed`
- Compare the MD5 checksum of the loaded code with that of the Beam file
- Erlang Shell: `mm()` and `lm()`
- Code contributed by Klarna, available since OTP 20

# The Code Primitives: `erlang:*(...)`

- Low-level primitives
    - No knowledge about search paths, stickiness, applications, etc.
    - Code server is built on top of these
- Only use if you know what you’re doing
- Can make the code server lose track of what’s loaded and how
- Faster (no server process communication overhead)

# Some useful primitives

- `erlang:loaded()`
  - List all loaded modules (without asking the code server)
- `erlang:module_loaded(ModName) -> true | false`
  - Check if the *Current* slot is populated
- `erlang:check_old_code(ModName) -> true | false`
  - Check if the *Old* slot is populated
- `erlang:check_process_code(Pid, Module)`
  - Check if Pid is referencing Module (expensive; can be used asynchronously)
- `erlang:pre_loaded()`
  - List all modules preloaded into the Beam as part of bootstrapping

# Atomic load of multiple modules

- New since OTP 19
  - We had been hinting to the OTP team for some years that this would be a good thing
- All-or-nothing load of multiple modules as a single operation
- Avoids certain race conditions between modules
  - Newly loaded version of module n wants to call new function foo/3 in module m, but the new version of m has not been loaded yet, so the call fails in mid-upgrade
- `code:atomic_load(ListOfModules)`
  - Can fail, e.g. if a module contains an `on_load` directive; if so, nothing is changed
- Two-phase loading:
  - `code:prepare_loading(Modules)` - does most of the hard work, but no visible effects
  - `code:finish_loading(Prepared)` - atomically switches to the new versions

# Beam code and Erlang VM compatibility

- Generally, you are guaranteed to be able to take Beam code compiled using a particular Erlang version and run it on the VM of the next major release.
- This has sometimes not been true
    - Incompatibility of Fun representations between OTP 16 and 17
    - Dropped support for “tuple calls” in OTP 21 (now fixed by a compatibility flag)

# Mindset

# Expect failure now and then

- Do your best to get upgrades right
- But don’t trust that they will not go wrong occasionally
- Write resilient code that survives hiccups
- Logging is your friend

# Traditional software code scenarios

- Cold start, empty system
  - No compatibility worries for the local data
- Restart, populated system
  - Persistent state (files) could be using older formats
- Other machines may be using older protocols

# Live upgrade code scenarios

- All the traditional scenarios, plus:
- Old data formats still in use, in RAM
  - ETS tables
  - Message queues
  - Process states
- Knowing when it’s safe to remove code handling old formats
- Might need upgrade actions to e.g. rewrite stuff in ETS tables.

# Beware systematic failure on all nodes

- Can be hard to avoid upgrading all nodes at once
  - They need to cooperate about any new feature being added
- If a critical component crashes on all nodes, you’re down
  - They may all hit the same new bug within a short time span
- Feature switches may enable new code on some nodes only

# There is no Rollback - Only Roll Forward

- Any nontrivial change could have affected the system as soon as it went live
  - Runtime state of processes, arguments being passed around
  - Messages between processes, in flight or in queues
  - Data written to database tables, files
- Rolling back the code makes it incapable of handling the new data
- Push a new change that fixes the problem
  - Maybe a quick fix first to paper over the worst effects, followed by a full fix later
- No real point in trying to make your toolchain support rollback

# Techniques

# Sanity Checks

- Xref checks to ensure new version has no missing functions
- Anything else specific to your system and upgrades

# Separate releasing from activating

- Feature switches
  - Ship the code but let it do nothing until activated
- Release in multiple phases
  - Ship code that handles new data formats before the code that produces it
  - Handle both versions for a transition period
  - Drop the obsolete code

# Separate may-crash from must-not-crash

- Many subsystems/applications can be allowed to crash and restart
  - Use normal precautions for upgrades, but assume things can go wrong
  - If they keep crashing, you can apply a quick fix
- Some subsystems/applications are critical to the node
  - Keep the code simple and obvious - put complex stuff somewhere separate
  - Be very conservative with changes
  - If there is a failure, it’s usually better if the whole node restarts

# Temporary Dirty Hacks

- Exporting functions that are needed for some specific upgrade action
  - Unexport again or make official API
- Using the sys module and other internal functions in OTP to modify the state of servers and supervisors on the fly
- Putting soft links on disk to handle file paths that need changing

# Full upgrade testing

- Part of CI pipeline, typically in Docker
- Small cluster, not just one machine
- Simulated load, not just idling
- Set up and start current live version, warm up
- Deploy candidate version and perform upgrade
- Collect error logs, metrics

# Pitfalls

# Remember to enable code migration

- Any service or long-running task should regularly do a qualified tail call
- If it does not, it will be stuck in the old code until done or killed
- Can be easy to miss even in a code review

# Remember to update settings everywhere

- Upgrades often need an action to modify a setting
- This must be done both to the running system and on disk
- Easy to forget to make persistent so you lose it if the node restarts

# Summary

# Advantages and disadvantages

+ Small impact on running system - release often
+ Fix problems or add features without stopping anything
+ Gradually reshape the code
- Needs careful coding, knowledge, code reviews
- Some changes must be released in phases

# Should you be doing it?

- Yes, if not having to stop nodes is a major advantage to you
- Might not be worth it if you’re happy with stopping nodes
  - Maybe only for quick fixes

# The End

# Load Your Code on the Road