Python @payments @scale

whoami

- Paweł Królikowski
  - rabbbit@gmail.com
- Writing Python for ~6 years
- Experience in:
  - Live sport data processing
  - Flask/REST
  - Recently payments collection

Python @payments @scale

- A story, rather than a technical deep-dive

Python @payments @scale

- A story, rather than a technical deep-dive
    - A company growing exponentially, outgrowing the solutions and assumptions
    - 50%+ was Python

Python @payments @scale

- A story, rather than a technical deep-dive

    - A company growing exponentially, outgrowing the solutions and assumptions
    - 50%+ was Python
    - Will briefly touch upon all aspects development
    - Relatively quickly paced, without too many technical details

Python @payments @scale

- A story, rather than a technical deep-dive.
    - A company growing exponentially, outgrowing the solutions and assumptions
    - 50%+ was Python
    - Will briefly touch upon all aspects development
    - Relatively quickly paced, without too many technical details

- Spoiler - not a happy ending :)

Plan

- Background
- History
  - Bad & Cool things
- Current landscape
  - General
    - Development + Infrastructure
    - Python tooling
  - Payments specific
- Problems
  - “Planet-scale” issues
  - Payments specific
- Conclusion

Background

- Currently:
  - 2294+ active micro-services, 500+ Python, 400+ Node, 500+ Go, 200+ Java

Background

- Currently:
  - 2294+ active micro-services, 500+ Python, 400+ Node, 500+ Go, 200+ Java
  - 3000+ engineers

Background

- Currently:
  - 2294+ active micro-services, 500+ Python, 400+ Node, 500+ Go, 200+ Java
  - 3000+ engineers

- Started in 2011-2012 - explosive growth, in all dimensions

Background

- Currently:
  - 2294+ active micro-services, 500+ Python, 400+ Node, 500+ Go, 200+ Java
  - 3000+ engineers

- Started in 2011-2012 - explosive growth, in all dimensions
  - Example: Mid 2015: 1000 engineers

Background

- Currently:
  - 2294+ active micro-services, 500+ Python, 400+ Node, 500+ Go, 200+ Java
  - 3000+ engineers

- Started in 2011-2012 - explosive growth, in all dimensions
  - Example: Mid 2015: 1000 engineers
  - A lot of things were moving very, very fast - *technical debt is a choice*

Background

- Currently:
  - 2294+ active micro-services, 500+ Python, 400+ Node, 500+ Go, 200+ Java
  - 3000+ engineers

- Started in 2011-2012 - explosive growth, in all dimensions
  - Example: Mid 2015: 1000 engineers
  - A lot of things were moving very, very fast - *technical debt is a choice*
  - All the assumptions turned out to be incomplete, multiple times - average service life-time is 9-12 months

History

- Started with PHP

History

- Started with PHP

- Python started in 2011, Node soon after
  - Node was responsible for the “realtime” part of the system - requesting rides, matching drivers and riders, ...
  - Python was responsible for the “backend”: onboarding, communication, reporting, payments, ...

History

- Started with PHP

- Python started in 2011, Node soon after
  - Node was responsible for the “realtime” part of the system - requesting rides, matching drivers and riders, ...
  - Python was responsible for the “backend”: onboarding, communication, reporting, payments, ...

- The first Python service was “The API”
  - Like most startups, start with a single codebase, start splitting things up
  - In Pyramid, a few other Flask services followed soon

History - The API

```
git log --oneline | wc -l
   39567
```

```
Date:   Tue Jan 25 16:16:22 2011 -0800

    Initial Commit
```

```
Date:   Tue Mar 7 16:25:49 2017 -0800

    [Very Low Risk] [Darwin Blocker] Delete the API
```

History - The API

History - The API

- “Darwin” started late/mid 2014
- Late 2015 at 200K LOC (not tests), expected 40+ services, 24+ teams
- 3000+ machines

History - The API

- “Darwin” started late/mid 2014
- Late 2015 at 200K LOC (not tests), expected 40+ services, 24+ teams
- 3000+ machines

- “Noisy neighbours” - your change can bring down everything
  - uWSGI worker starvation, shared Celery queues

History - The API

- “Darwin” started late/mid 2014
- Late 2015 at 200K LOC (not tests), expected 40+ services, 24+ teams
- 3000+ machines

- “Noisy neighbours” - your change can bring down everything
  - uWSGI worker starvation, shared Celery queues
- Tens of deployments per day
  - Scheduled deployers, tools just for managing the load

History - The API

- “Darwin” started late/mid 2014
- Late 2015 at 200K LOC (not tests), expected 40+ services, 24+ teams
- 3000+ machines

- “Noisy neighbours” - your change can bring down everything
  - uWSGI worker starvation, shared Celery queues
- Tens of deployments per day
  - Scheduled deployers, tools just for managing the load
- Very difficult on-call

History - The API

- “Darwin” started late/mid 2014
- Late 2015 at 200K LOC (not tests), expected 40+ services, 24+ teams
- 3000+ machines

- “Noisy neighbours” - your change can bring down everything
  - uWSGI worker starvation, shared Celery queues
- Tens of deployments per day
  - Scheduled deployers, tools just for managing the load
- Very difficult on-call

- It worked!

History - The API

- “Darwin” started late/mid 2014
- Late 2015 at 200K LOC (not tests), expected 40+ services, 24+ teams
- 3000+ machines

- “Noisy neighbours” - your change can bring down everything
  - uWSGI worker starvation, shared Celery queues
- Tens of deployments per day
  - Scheduled deployers, tools just for managing the load
- Very difficult on-call

- Never intended to be that big - example: “418: I’m a teapot”
  - And it’s here to stay

History - The API

- “Darwin” started late/mid 2014
- Late 2015 at 200K LOC (not tests), expected 40+ services, 24+ teams
- 3000+ machines

- “Noisy neighbours” - your change can bring down everything
  - uWSGI worker starvation, shared Celery queues
- Tens of deployments per day
  - Scheduled deployers, tools just for managing the load
- Very difficult on-call

- Never intended to be that big - example: “418: I’m a teapot”

History - The API - Cool things

- Tests: sharded, running on Jenkins in 6 minutes, no flakiness

History - The API - Cool things

- Tests: sharded, running on Jenkins in 6 minutes, no flakiness

- Dynamic, memory mapped translations

History - The API - Cool things

- Tests: sharded, running on Jenkins in 6 minutes, no flakiness
- Dynamic, memory mapped translations
- Rolling deploys/restarts in 5 minutes (!!!)

History - The API - Cool things

- Tests: sharded, running on Jenkins in 6 minutes, no flakiness
- Dynamic, memory mapped translations
- Rolling deploys/restarts in 5 minutes (!!!)
- Clay - Libraries for dynamic configuration, stats, master/slave selection, environment

History - The API - Cool things

- Tests: sharded, running on Jenkins in 6 minutes, no flakiness
- Dynamic, memory mapped translations
- Rolling deploys/restarts in 5 minutes (!!!)
- Clay - Libraries for dynamic configuration, stats, master/slave selection, environment
- SqlAlchemy + Alembic

History - The API - Celery

- As much possible ran through async queues

History - The API - Celery

- As much possible ran through async queues
- Multi-master, with high/low priority tasks

History - The API - Celery

- As much possible ran through async queues
- Multi-master, with high/low priority tasks
- Automatic alerts/task priority allocation

History - The API - Celery

- As much possible ran through async queues
- Multi-master, with high/low priority tasks
- Automatic alerts/task priority allocation
- Quarantine queues (!)

History - The API - Celery

- As much possible ran through async queues
- Multi-master, with high/low priority tasks
- Automatic alerts/task priority allocation
- Quarantine queues (!)
- Serious outages avoided/reduced

History - post API

- With microservices, heavily IO Bound, async was necessary
- Painful migration - code path by code path had to be re-written
- The code was surprisingly complicated
- Architecture was surprisingly complicated too

History - post API

- With microservices, heavily IO Bound, async was necessary

History - post API

- With microservices, heavily IO Bound, async was necessary
- Painful migration - code path by code path had to be re-written

History - post API

- With microservices, heavily IO Bound, async was necessary
- Painful migration - code path by code path had to be re-written
- The code was surprisingly complicated

History - post API

- With microservices, heavily IO Bound, async was necessary
- Painful migration - code path by code path had to be re-written
- The code was surprisingly complicated
- Architecture was surprisingly complicated too

Current landscape

Current landscape - overview

- Tornado, everyone free to pick version, teams are independent
  - There’s some flask running still

Current landscape - overview

- Tornado, everyone free to pick version, teams are independent
  - There’s some flask running still
- Python 2.7.12 :(
  - Python3 in adhoc scripts

Current landscape - overview

- Tornado, everyone free to pick version, teams are independent
  - There’s some flask running still
- Python 2.7.12 :(
  - Python3 in adhoc scripts
- Heavily language agnostic
  - Services talk Thrift
  - Automatic service discovery

Current landscape - development

- Largely language agnostic too

Current landscape - development

- Largely language agnostic too

- 80%+ test coverage
- Mandatory code reviews

Current landscape - development

- Largely language agnostic too

- 80%+ test coverage
- Mandatory code reviews
- One-click service set-up

Current landscape - development

- Largely language agnostic too

- 80%+ test coverage
- Mandatory code reviews
- One-click service set-up
- “Testing in production”

Current landscape - development

- Largely language agnostic too

- 80%+ test coverage
- Mandatory code reviews
- One-click service set-up
- “Testing in production”
- Load/integration/disaster testing

Current landscape - infrastructure

- Targeted at empowering each teams

Current landscape - infrastructure

- Targeted at empowering each teams

- One click deployment
- Staged rollouts

Current landscape - infrastructure

- Targeted at empowering each teams

- One click deployment
- Staged rollouts
- Gated rollouts (integration tests)

Current landscape - infrastructure

- Targeted at empowering each teams

- One click deployment
- Staged rollouts
- Gated rollouts (integration tests)
- Docker

Current landscape - infrastructure

- Targeted at empowering each teams

- One click deployment
- Staged rollouts
- Gated rollouts (integration tests)
- Docker
- Monitoring

Current landscape - infrastructure

- Targeted at empowering each teams

- One click deployment
- Staged rollouts
- Gated rollouts (integration tests)
- Docker
- Monitoring
- Dynamic configuration, translations, experiments

Current landscape - infrastructure

- Targeted at empowering each teams

- One click deployment
- Staged rollouts
- Gated rollouts (integration tests)
- Docker
- Monitoring
- Dynamic configuration, translations, experiments
- Distributed tracing

Current landscape - Python infrastructure

- PyPI

Current landscape - Python infrastructure

- PyPI

- Pypy - tried, failed

Current landscape - Python infrastructure

- PyPI
- Pypy - tried, failed
- Profiling: pyflames, pympler, plop - guidelines, as well as designated team

Current landscape - Python infrastructure

- PyPI
- Pypy - tried, failed
- Profiling: pyflames, pympler, plop - guidelines, as well as designated team
- Cython, uvloop - applied selectively

Current landscape - Python infrastructure

- PyPI
- Pypy - tried, failed
- Profiling: pyflames, pympler, plop - guidelines, as well as designated team
- Cython, uvloop - applied selectively
- Jupyter notebooks

Payments

Python at Payments

- All of payments used to be in Python
  - 5b trips, 10+m trips a day
  - Up until ~6 months ago

Python at Payments

- All of payments used to be in Python
  - 5b trips, 10+m trips a day
  - Up until ~6 months ago

- In particular, rider collection
  - ~10 Tornado services
  - 200-300k LOC
  - 100k LOC library that deals with 30+ integrations (Paypal, etc)

Python at Payments

- Python matched well
  - Relatively low write throughput (~1000RPS)
  - Relatively low read throughput (~20-30k RPS) / service
  - Integrations, in particular, we mostly data manipulation/parsing - Python is convenient

Python at Payments

- Python matched well
  - Relatively low write throughput (~1000RPS)
  - Relatively low read throughput (~20-30k RPS) / service
  - Integrations, in particular, we mostly data manipulation/parsing - Python is convenient

- SLOs are relaxed
  - Difference between 10 and 60 seconds (usually) irrelevant
  - Vast majority of actions asynchronous
  - Consistent, not fast

Python at Payments

- Python matched well
  - Relatively low write throughput (~1000RPS)
  - Relatively low read throughput (~20-30k RPS) / service
  - Integrations, in particular, we mostly data manipulation/parsing - Python is convenient
- SLOs are relaxed
  - Difference between 10 and 60 seconds (usually) irrelevant
  - Vast majority of actions asynchronous
  - Consistent, not fast
- Fun fact: on internal errors, we fail-open

Problems

Problems - at “planetscale”

- Performance
  - it is simply slower: it doesn’t matter under a certain scale, but once you hit 10k+ RPS it’s a problem

Problems - at “planetscale”

- Performance
  - it is simply slower: it doesn’t matter under a certain scale, but once you hit 10k+ RPS it’s a problem
  - Money runs several services, 25k+ RPS, ~600 instances, 1200 cores, at 250% headroom.

Problems - at “planetscale”

- Performance
  - it is simply slower: it doesn’t matter under a certain scale, but once you hit 10k+ RPS it’s a problem.
  - Money runs several services, 25k+ RPS, ~600 instances, 1200 cores, at 250% headroom.
  - Other teams 100k+ RPS.

Problems - at “planetscale”

- Performance
  - it is simply slower: it doesn’t matter under a certain scale, but once you hit 10k+ RPS it’s a problem.
  - Money runs several services, 25k+ RPS, ~600 instances, 1200 cores, at 250% headroom.
  - Other teams 100k+ RPS.

- Battlestories:

Problems - at “planetscale”

- Performance
  - it is simply slower: it doesn’t matter under a certain scale, but once you hit 10k+ RPS it’s a problem.
  - Money runs several services, 25k+ RPS, ~600 instances, 1200 cores, at 250% headroom.
  - Other teams 100k+ RPS.

- Battlestories:
  - json -> ujson

Problems - at “planetscale”

- Performance
  - it is simply slower: it doesn’t matter under a certain scale, but once you hit 10k+ RPS it’s a problem.
  - Money runs several services, 25k+ RPS, ~600 instances, 1200 cores, at 250% headroom.
  - Other teams 100k+ RPS.

- Battlestories:
  - json -> ujson
  - Iso8601 -> ciso8601

Problems - at “planetscale”

- Performance
  - it is simply slower: it doesn’t matter under a certain scale, but once you hit 10k+ RPS it’s a problem.
  - Money runs several services, 25k+ RPS, ~600 instances, 1200 cores, at 250% headroom.
  - Other teams 100k+ RPS.

- Battlestories:
  - json -> ujson
  - Iso8601 -> ciso8601
  - serialisation/deserialisation: schematics, object hydration (thrift+database)

Problems - at “planetscale”

- Performance
  - it is simply slower: it doesn’t matter under a certain scale, but once you hit 10k+ RPS it’s a problem.
  - Money runs several services, 25k+ RPS, ~600 instances, 1200 cores, at 250% headroom.
  - Other teams 100k+ RPS.

- Battlestories:
  - json -> ujson
  - Iso8601 -> ciso8601
  - serialisation/deserialisation: schematics, object hydration (thrift+database)
  - Snake_case -> CamelCase conversion

Problems - complexity

- Tornado & Python2 are separate languages

Problems - complexity

- Tornado & Python2 are separate languages
- Static Typing

Problems - complexity

- Tornado & Python2 are separate languages
- Static Typing
- “Maturity”
  - “Missing mysql driver”
  - Profiling tools in production

Problems - complexity

- Tornado & Python2 are separate languages
- Static Typing
- “Maturity”
  - “Missing mysql driver”
  - Profiling tools in production
- Debugging
  - Stack traces

Problems - complexity

- Tornado & Python2 are separate languages
- Static Typing
- “Maturity”
  - “Missing mysql driver”
  - Profiling tools in production
- Debugging
  - Stack traces
- Excellent for productivity, expressiveness
  - worked great when Uber was a smaller company. “Now this flexibility works against us”.

Problems - 3rd choice

- Support burden of third language was too big

Problems - payments specific

- Type safety

Problems - payments specific

- Type safety

- Interface creep

Problems - payments specific

- Type safety
- Interface creep
- Error handling

Problems - payments specific

- Type safety
- Interface creep
- Error handling
- Java better for audits, more money-related libraries

Problems - payments specific

- Type safety
- Interface creep
- Error handling
- Java better for audits, more money-related libraries
- All of the previous
  - Debugging in prod - memory leaks, performance at times

Problems - payments specific

- Type safety
- Interface creep
- Error handling
- Java better for audits, more money-related libraries
- All of the previous
  - Debugging in prod - memory leaks, performance at times
  - Flexibility is too dangerous

Problems - conclusion

- Python is now deprecated

Recap

Recap

- Despite of how I made it sound like, Python is awesome

Recap

- Despite of how I made it sound like, Python is awesome

- Really!
  - Brought Uber where it is, allowed for quick iteration/growth
  - Extremely flexible, rich eco system
  - Would still use for smaller projects, scripting

Recap

- Despite of how I made it sound like, Python is awesome

- Really!
  - Brought Uber where it is, allowed for quick iteration/growth
  - Extremely flexible, rich eco system
  - Would still use for smaller projects, scripting

- Main problem - ease of long-term development (at scale)
  - The flexibility is easy to abuse
  - Static typing makes life easier (is serverless the future?)

Thank you!

Questions?

(rabbbit@gmail.com, pawel@uber.com)