From a monolith to a microservice shop architecture

Thomas Bley

About me

- Senior PHP Developer
- Linux, PHP, MySQL since 2001
- studied at TU München
- working for Bringmeister in Berlin

The Monolith

- Magento 1 Enterprise: support was discontinued by Adobe, license costs
- Large codebase with 1.8 mloc, developed over 10y years by 130 developers:
  - high complexity, strong coupling, multiple API layers
  - unused / unknown / bad code, hard to maintain, bugs
  - security issues, not designed for GDPR
  - many frameworks: Magento Varien, Zend Framework 1, Symfony 2, Laravel 5, ReactPHP, node.js
  - no Composer
- Performance issues:
  - add product to cart 400 queries, order placement 3.2k queries (5 transactions)
  - slow queries, deadlocks, memory limit issues
  - Admin login >60s, single order search in Admin 15s
  - database designed by Entity–attribute–value model (big joins)
  - aggregation tables filled synchronously

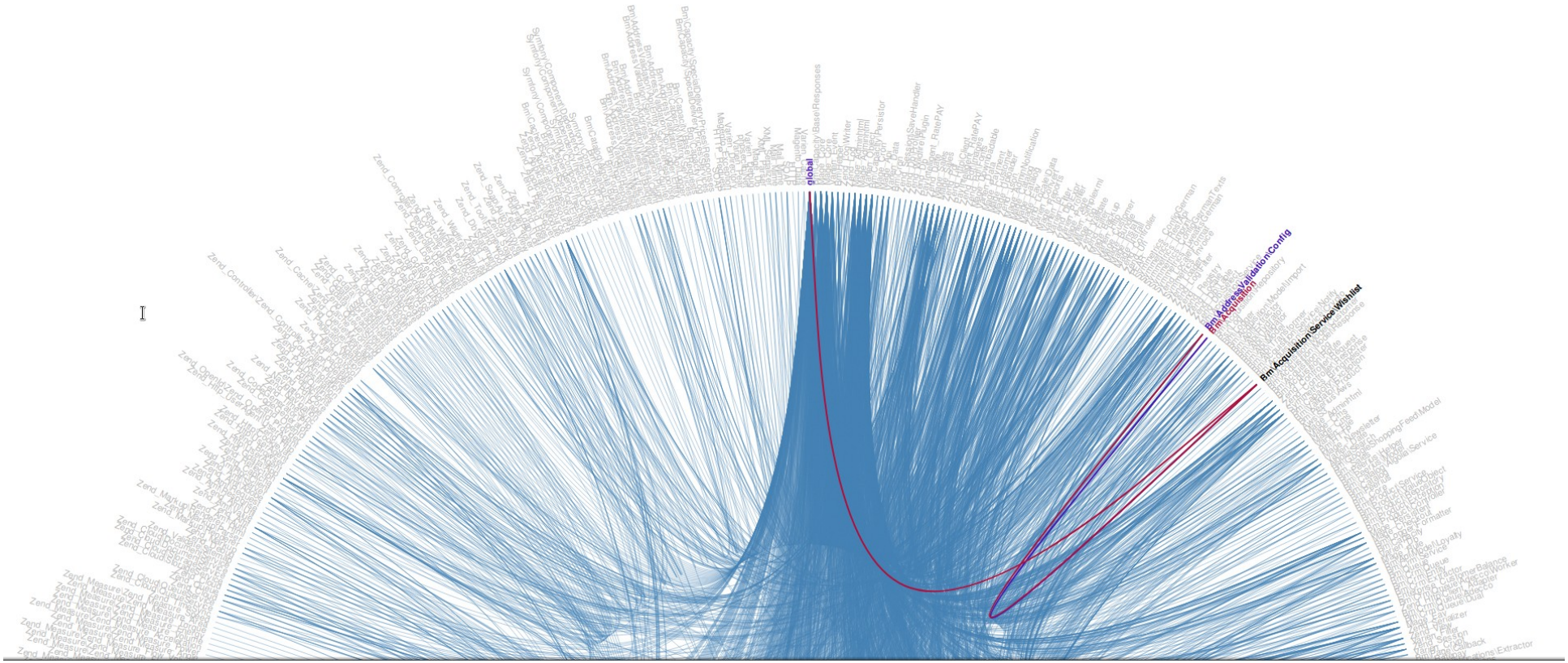
  ➔ difficult to scale the business

The Monolith #2

- Tests:
  - coverage <10%, require 7 GB database dump
  - many tests broken, only run locally, CI only used for deployment
  - mostly manual testing
- Productivity:
  - slow development of new features
  - analyzing production issues very complex
  - updates only applied manually
- Multiple sources for product data, mostly not in sync:
  - Database
  - Solr, Algolia
- Small team: 5 developers

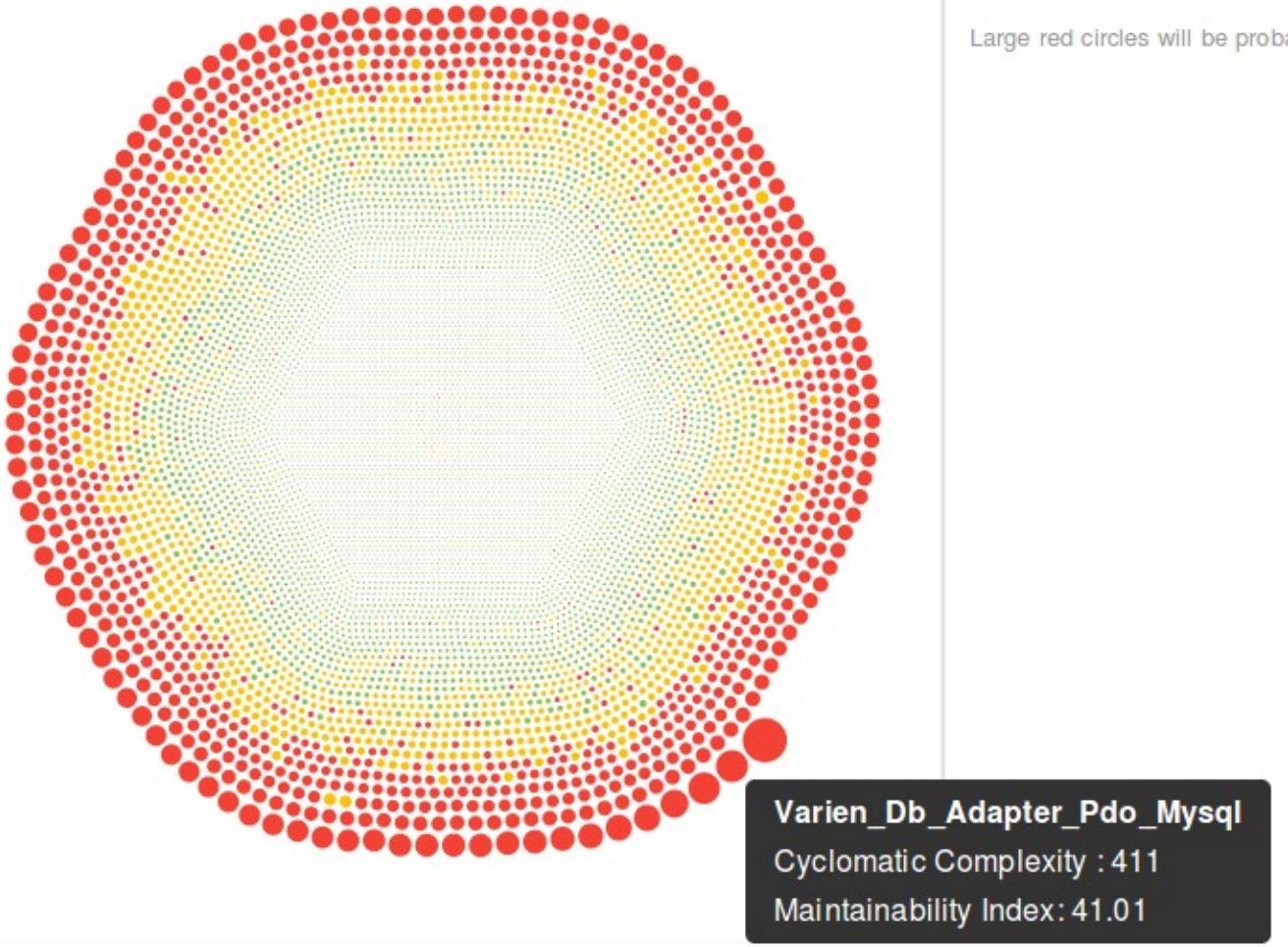
Developers not happy, Management not happy, Customers not happy with the shop

How does the monolith look like?



How complex is it?

Maintainability / complexity



Each file is symbolized by a circle. Size of the circle represents the Cyclomatic complexity. Color of the circle represents the Maintainability Index.

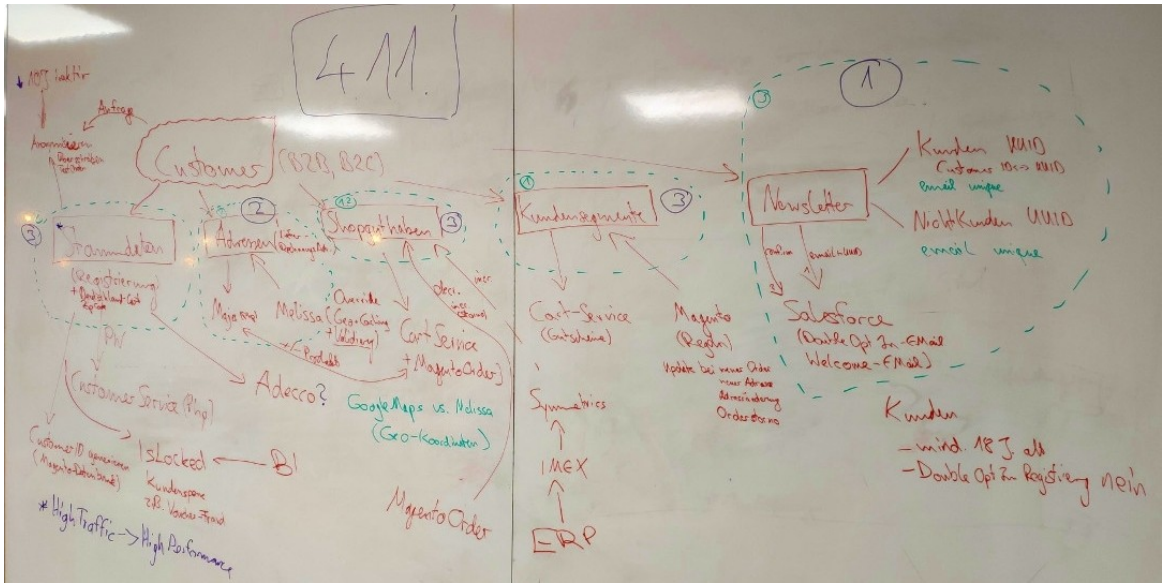
Large red circles will be probably hard to maintain.

CI/CD, Operations

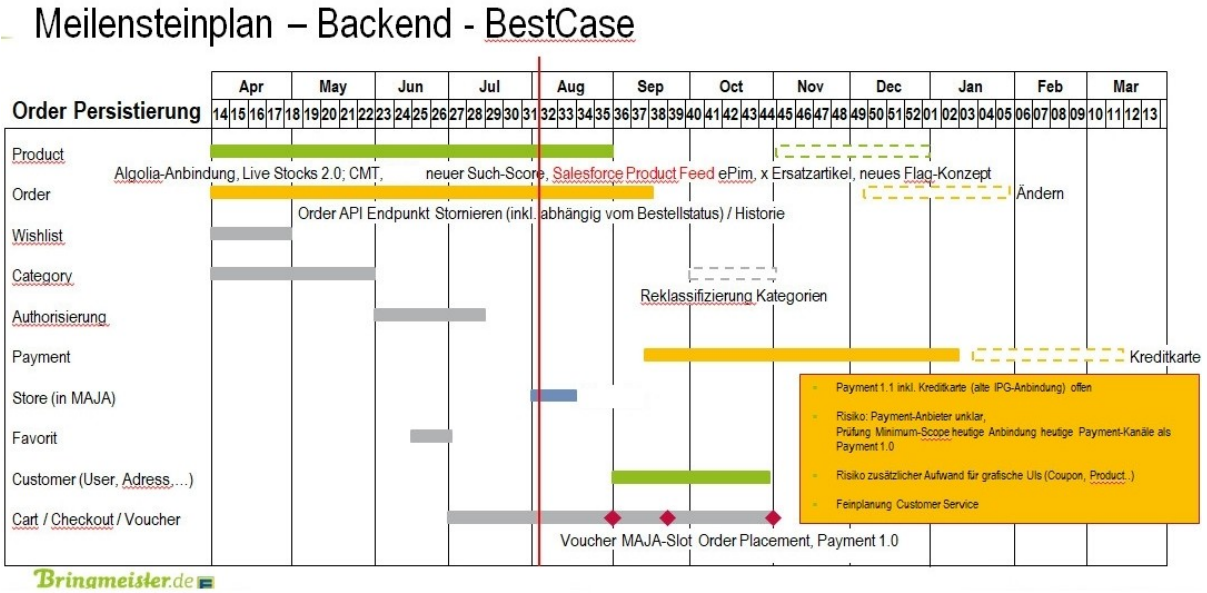
Why microservices?

- Smaller code base to work on
  - easier to develop, easier to change, easier to maintain
  - less complexity, focus on problem solving
- More options for databases and programming languages
- Easier to split work on multiple teams
- Hard system boundary with standard communication between services
- Limit impact of bugs and failures
- Isolation of data
  - less complexity in data storage
  - more complexity to join data

Explaining the project to developers



Explaining the project to management



Define the future architecture

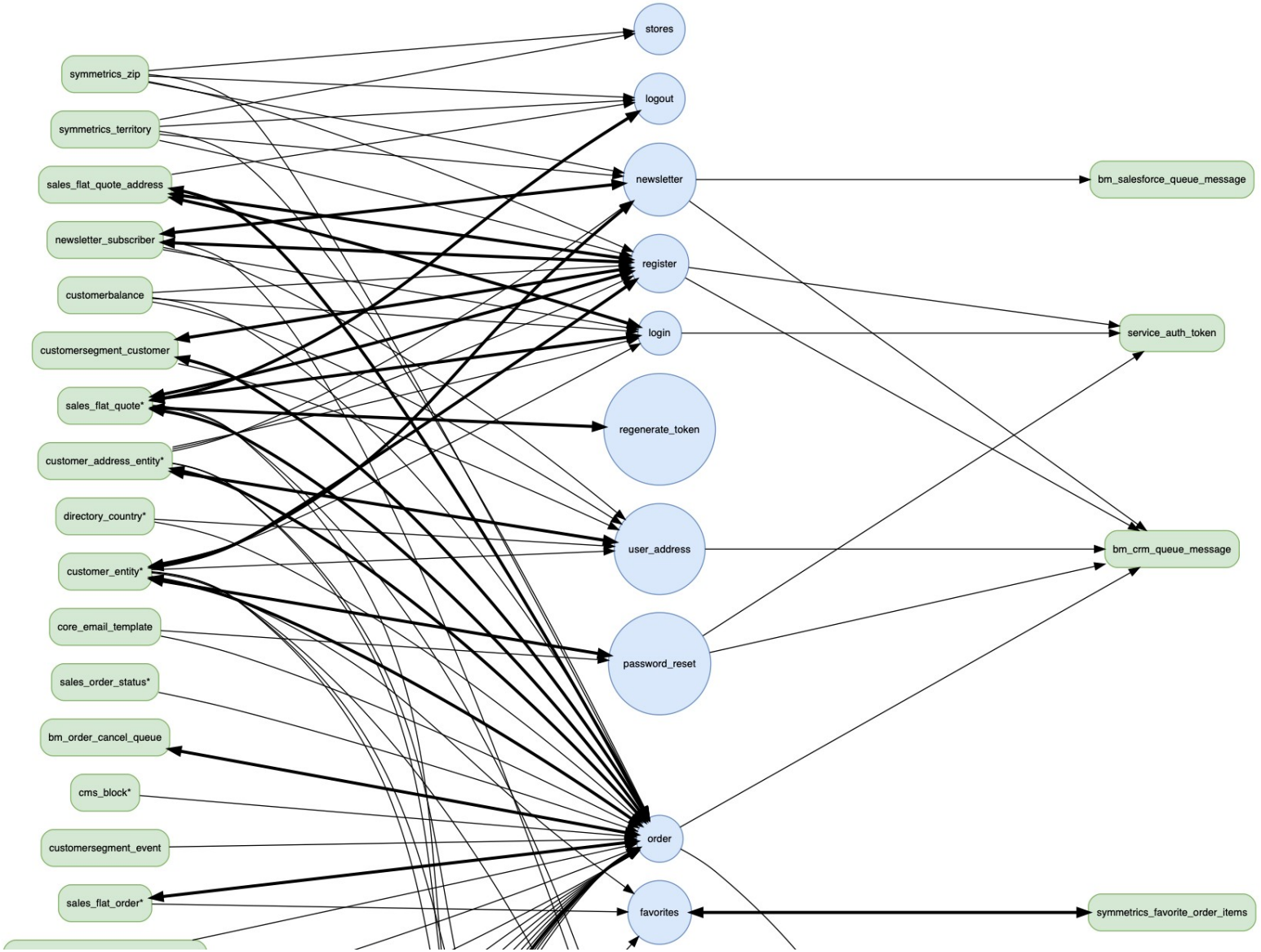
- We decided to stay with PHP and MySQL

  most experience, all required libraries and SDKs available, easier to port from PHP to PHP
- Keep existing server infrastructure
- No fullstack framework
  ➔ use our own mini-framework (200 loc)
- Write queries and schema definitions directly in SQL, no ORM, reduce joins by using JSON columns, no foreign keys
- Use JWT instead of sessions
- Single Monorepo for all services, each service with own code base, own database, own composer.json, etc.
- All product data in Algolia
- SOLID, Kiss

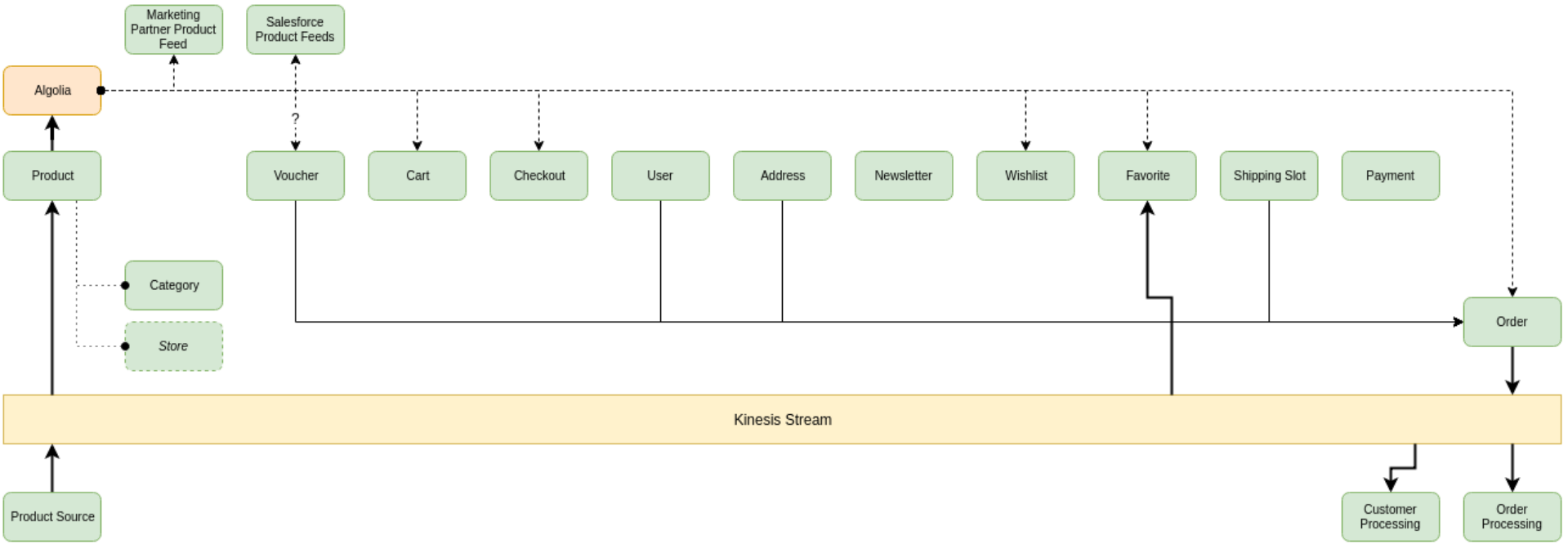
source: https://www.youtube.com/watch?v=fCt2_AsCWKI

- 100% test coverage, keep Behat tests
- Static code analysis with Psalm
- Enforce coding styles with PHP-CS-Fixer
- new CI/CD with Bitbucket Pipelines

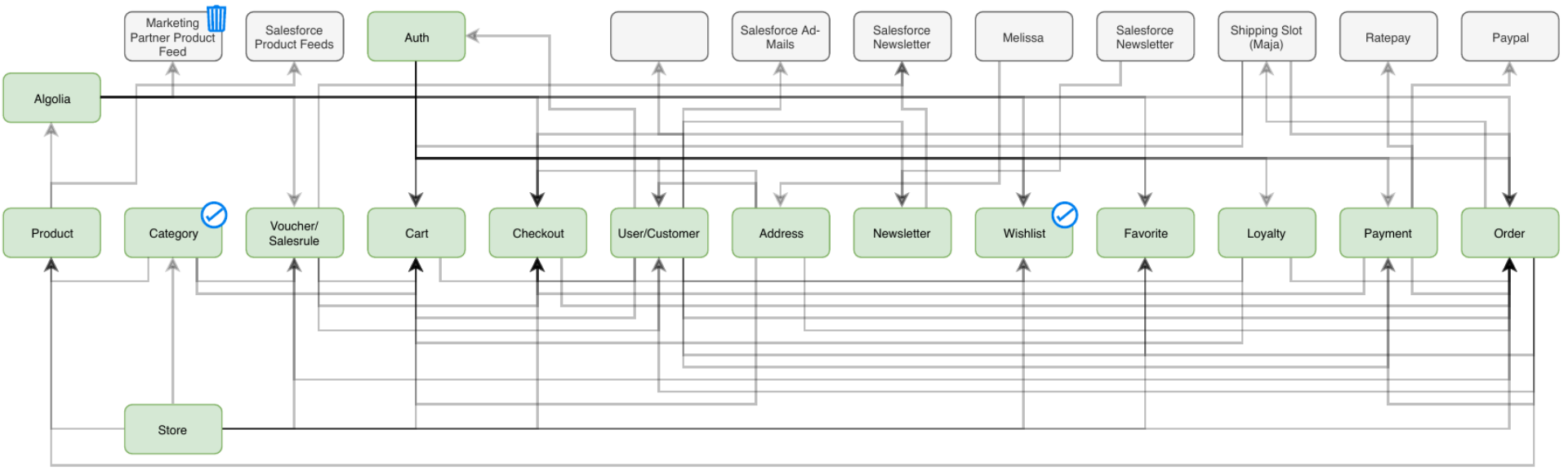
Analyze dependencies on database level



Identify services

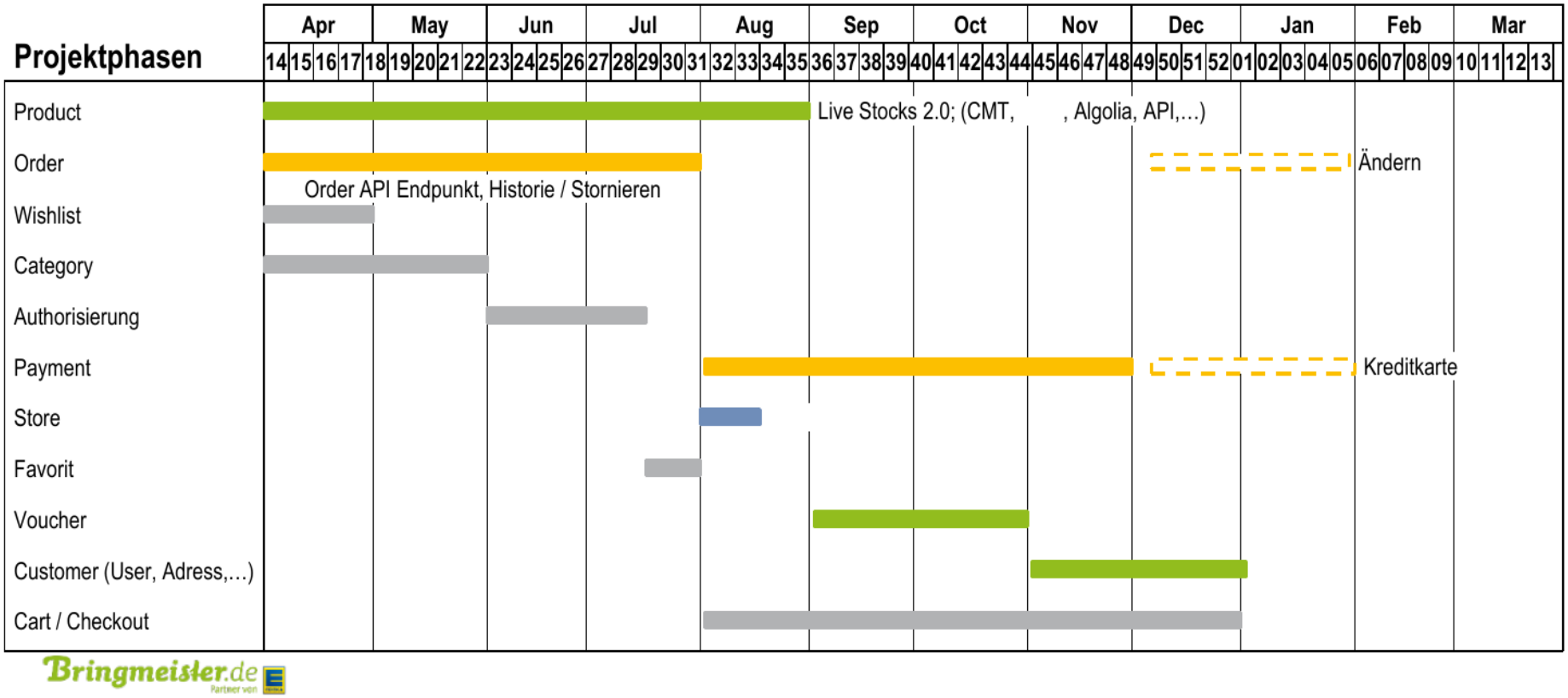


Start with easy services



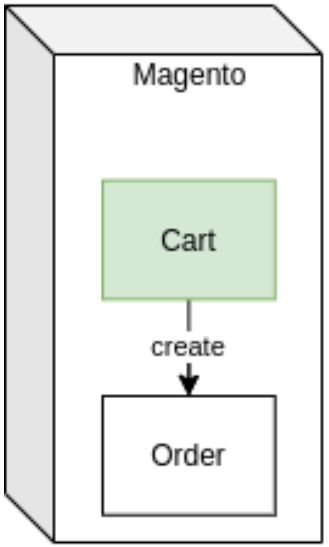
Implement and launch one by one

Meilensteinplan – Backend - BestCase

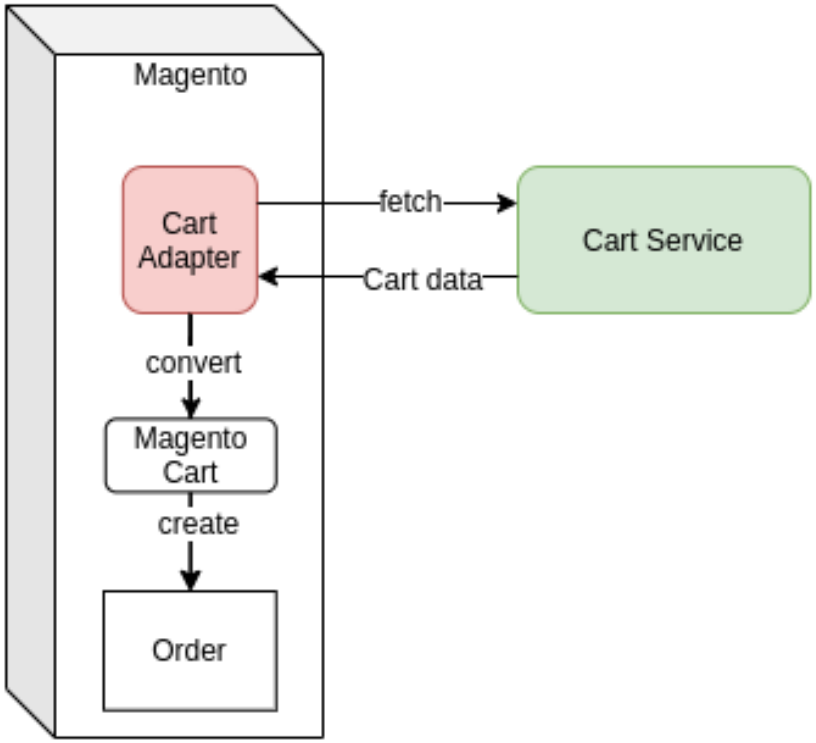


Build adapters to cut out components with strong dependencies

Before:



After:



Communication between Microservices and external providers

- Synchronous using REST, load balancer (strong consistency)
  - Customer master data, addresses
  - Cart, Vouchers
  - Product data
  - Payment providers
- Synchronous using SOAP (strong consistency)
  - Legacy systems (tour planning)
- Asynchronous using Events (Kinesis)
  - Orders, Logging
- Asynchronous using REST and queues
  - external providers (CRM, Customer Support systems, etc.)
- Forward customer’s JWT token between services

Data migration (EAV → JSON)

Using SQL to copy data from old schema to new schema:

```
INSERT INTO customers.customer (email, attributes, created)
    SELECT * FROM magento.customer_entity
    LEFT JOIN magento.customer_entity_varchar as firstname on
        firstname.entity_id = customer_entity.entity_id and firstname.attribute_id = 5
    LEFT JOIN magento.customer_entity_varchar as lastname on
        lastname.entity_id = customer_entity.entity_id and lastname.attribute_id = 7
    LEFT JOIN magento.customer_entity_datetime as birthdate on
        birthdate.entity_id = customer_entity.entity_id and birthdate.attribute_id = 11
    SET email = customer_entity.email,
        attributes = json_object(
            'first', firstname.value,
            'last', lastname.value,
            'birth', date(birthdate.value),
            'orders', (select count(*) from magento.sales_flat_order
                       where customer_id = customer_entity.entity_id)
        ),
        created = customer_entity.created_at;
```

Data anonymization

Using a separate database and views to provide anonymized production data:

```
SELECT * FROM customers.customer WHERE id = 1234;
    id: 1234
    email: foo.bar@baz.com
    attributes: {"first": "Thomas", "last": "Bley", "birth": "1930-02-01", "orders": 42}
    created: 2021-05-06 12:41:11

CREATE or REPLACE VIEW customers_anonymized.customer AS
    SELECT id,
        concat('invalid_', id, '@bringmeister.de') AS email,          # invalid_1234@bringmeister.de
        json_object(
            'first', concat('first_', id), 'last', concat('last_', id),          # first_1234, last_1234
            'birth', '1980-01-02', 'orders', attributes->>"$.orders"    # 1980-01-02, 42
        ) AS attributes,
        created
    FROM customers.customer;
```

Results

- Project finished in time and in quality (Mar - Nov 2019)
- 10 microservices, 3 admin interfaces
- 99.99% test coverage with unit and integration tests
- Code size reduced to 100 kloc (coming from 1.8 mloc)
- Data size in database reduced by 80%
- System performance and revenue significantly increased
- Hardware costs reduced by 50%
- External security audit passed
- Tests, Build and Deployment in < 10 minutes
- Development of new features and maintenance much quicker and easier

Developers happy, Management happy, Customers happy with the shop

Implement microservices like a monolith?

source: twitter.com/ddprrt/status/1425418538257428488

Learnings

- Smaller is better
- Don’t ignore or delay problems
- Requirements
- Documentation
- Rewriting is a lot of work, but it’s worth to do it
- Management support is important
- Monitoring (esp. external service providers, we use Datadog)
- Testing, testing, testing (PHPUnit, Psalm, Trivy, manual, etc.)
- Performance, performance, performance (build a 10x system)

Thanks for listening!

Questions?

download slides:
github.com/thomasbley/talks

follow me:
twitter.com/thbley