Software Architecture Reconstruction for Microservice Systems using Static Analysis via GraalVM Native Image

Richard Hutcheson [2], Austin Blanchard [2], Noah Lambaria [2], Jack Hale [2], **David Kozak** [3], Amr S. Abdelfattah [2], Tomas Cerny [1]

[1] University of Arizona
[2] Baylor University
[3] Brno University of Technology
ikozak@fit.vutbr.cz
d-kozak.github.io

March 13, 2024

Microservices commonly used for cloud-native systems

Microservices commonly used for cloud-native systems

Advantages:

- flexibility
- scalability
- facilitated deployment

Microservices commonly used for cloud-native systems

Advantages:

- flexibility
- scalability
- facilitated deployment

Disadvantages:

- distributed system
- complex
- error-prone

Microservices

Microservices commonly used for cloud-native systems

Advantages:

- flexibility
- scalability
- facilitated deployment

Disadvantages:

- distributed system
- complex
- error-prone

A holistic perspective is often **missing**.

- **Software Architecture Reconstruction (SAR)** for microservices generates high-level architectural views:

*https://www.iso.org/obp/ui/#iso:std:iso-iec-ieee:42010:ed-2:v1:en

- **Software Architecture Reconstruction (SAR)** for microservices generates high-level architectural views:
  - Views are common for describing software architecture*
  - Service view – interaction among services
  - Domain view – relations between database entities

*https://www.iso.org/obp/ui/#iso:std:iso-iec-ieee:42010:ed-2:v1:en

- **Software Architecture Reconstruction (SAR)** for microservices generates high-level architectural views:
  - Views are common for describing software architecture*
  - Service view – interaction among services
  - Domain view – relations between database entities

*https://www.iso.org/obp/ui/#iso:std:iso-iec-ieee:42010:ed-2:v1:en

- **Software Architecture Reconstruction (SAR)** for microservices generates high-level architectural views:
  - Views are common for describing software architecture*
  - Service view – interaction among services
  - Domain view – relations between database entities

- Provides a deeper understanding of the system:
  - Evolution, trade-off analysis, assigning responsibilities, etc

*https://www.iso.org/obp/ui/#iso:std:iso-iec-ieee:42010:ed-2:v1:en

- **Software Architecture Reconstruction (SAR)** for microservices generates high-level architectural views:
  - Views are common for describing software architecture*
  - Service view – interaction among services
  - Domain view – relations between database entities
- Provides a deeper understanding of the system:
  - Evolution, trade-off analysis, assigning responsibilities, etc
- Necessary precondition for detecting microservice smells:

*https://www.iso.org/obp/ui/#iso:std:iso-iec-ieee:42010:ed-2:v1:en

- **Software Architecture Reconstruction (SAR)** for microservices generates high-level architectural views:
  - Views are common for describing software architecture*
  - Service view – interaction among services
  - Domain view – relations between database entities
- Provides a deeper understanding of the system:
  - Evolution, trade-off analysis, assigning responsibilities, etc
- Necessary precondition for detecting microservice smells:
  - wrong cuts
  - shared persistency
  - cyclic dependencies

*https://www.iso.org/obp/ui/#iso:std:iso-iec-ieee:42010:ed-2:v1:en

Creating SAR Views

There are multiple ways how **SAR** views can be generated:

Creating SAR Views

There are multiple ways how **SAR** views can be generated:

- Manual Execution
  - Can be tedious and error-prone

Creating SAR Views

There are multiple ways how **SAR** views can be generated:

- Manual Execution
  - Can be tedious and error-prone
- Dynamic Analysis
  - Requires a runtime environment

Creating SAR Views

There are multiple ways how **SAR** views can be generated:

- Manual Execution
  - Can be tedious and error-prone
- Dynamic Analysis
  - Requires a runtime environment
- Static Analysis
  - Needs only the source code without execution
  - Traditional methods not suitable for SAR
  - Source code might not be available

Our Proposal

Perform **SAR** using static analysis on Java bytecode.

Our Proposal

Perform **SAR** using static analysis on Java bytecode.

- Fully automated
  - No room for manual error
  - Easy integration with a Continous Integration pipeline
  - Monitor architecture evolution over time

Our Proposal

Perform **SAR** using static analysis on Java bytecode.

- Fully automated
  - No room for manual error
  - Easy integration with a Continous Integration pipeline
  - Monitor architecture evolution over time
- No runtime environment necessary
  - Applicable as soon as the code is compiled

Our Proposal

Perform **SAR** using static analysis on Java bytecode.

- Fully automated
  - No room for manual error
  - Easy integration with a Continous Integration pipeline
  - Monitor architecture evolution over time
- No runtime environment necessary
  - Applicable as soon as the code is compiled
- Does not need source code
  - Can analyze libraries, third-party dependencies, and legacy code

GraalVM Native Image

- Ahead-of-time (AOT) compiler for Java
  - Fast startup time - from seconds to milliseconds
  - Low memory footprint

GraalVM Native Image

- Ahead-of-time (AOT) compiler for Java
  - Fast startup time - from seconds to milliseconds
  - Low memory footprint
- Highly popular in the industry
  - Support from all major frameworks including Spring, Micronaut, Quarkus, and Helidon

GraalVM Native Image

- Ahead-of-time (AOT) compiler for Java
  - Fast startup time - from seconds to milliseconds
  - Low memory footprint
- Highly popular in the industry
  - Support from all major frameworks including Spring, Micronaut, Quarkus, and Helidon
- Industrial-grade static analysis
  - To detect reachable program elements
  - We can reuse the domain classes and tools

- Graal Intermediate Representation (Graal IR)
  - Well-documented
  - Visualization tool Ideal Graph Visualizer
    - Inspect the IR
    - Detect high-level patterns

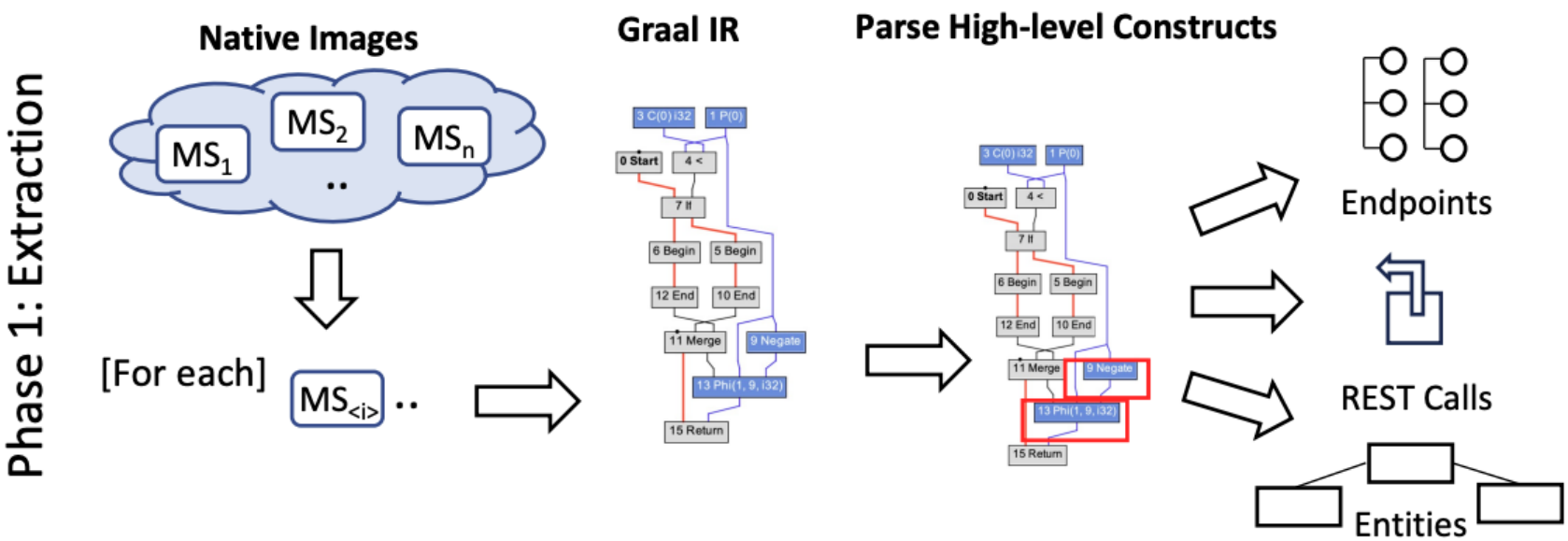
SAR using GraalVM Native Image

- Graal Intermediate Representation (Graal IR)
  - Well-documented
  - Visualization tool Ideal Graph Visualizer
    - Inspect the IR
    - Detect high-level patterns
- Open-source
  - We can modify it easily

Phase 1: Extraction

First, we process each microservice using annotation scanning and pattern matching on the IR to extract:

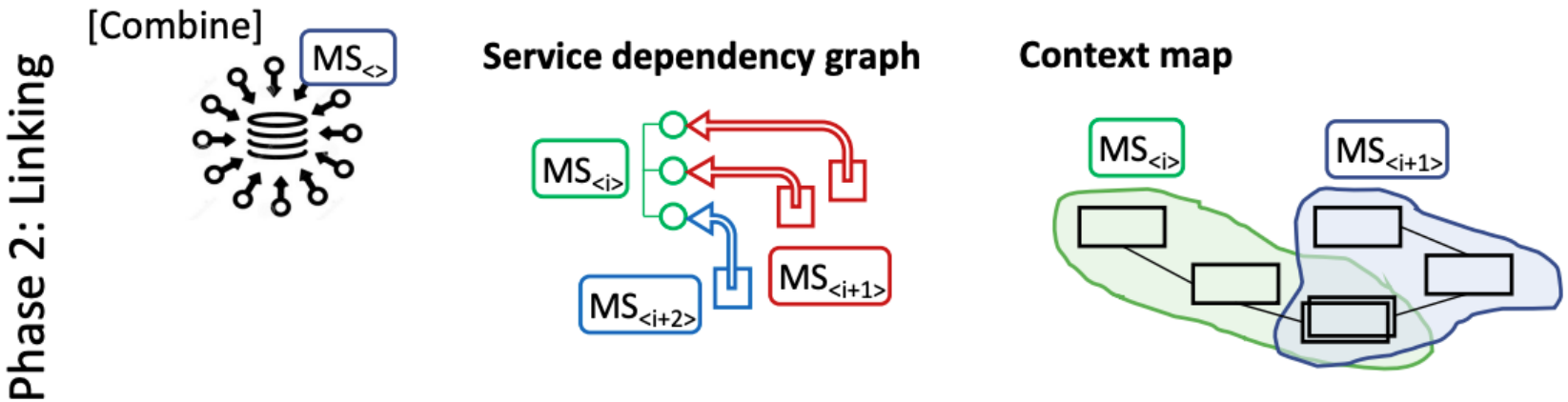
- Rest Endpoints
- Rest Calls
- Database Schema



Phase 2: Linking

Second, we combine the per-microservice domain data to generate SAR views:

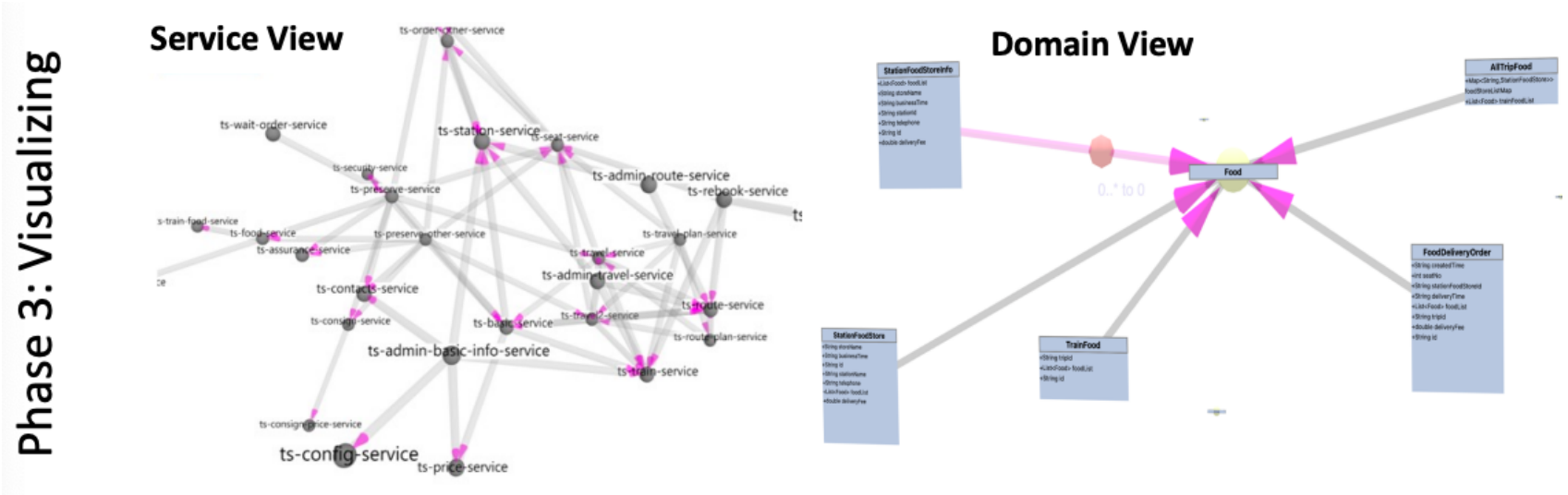
- Rest Calls linked with Rest Endpoints
- Database schemata merged based on equivalent entities between them



Phase 3: Visualization

Third, we visualize the obtained data to present a holistic view of the system:

- Service view – interaction among services
- Domain view – relations between database entities



Evaluation and Validation

We built a proof of concept - MicroGraal*:

- Tailored to JavaEE/Spring

*`https://github.com/cloudhubs/graal_mvp`

Evaluation and Validation

We built a proof of concept - MicroGraal*:

- Tailored to JavaEE/Spring

We valided our approach on **TrainTicket** v1.0.0:

*`https://github.com/cloudhubs/graal_mvp`

Evaluation and Validation

We built a proof of concept - MicroGraal*:

- Tailored to JavaEE/Spring

We valided our approach on **TrainTicket** v1.0.0:

- Well-established community benchmark

*`https://github.com/cloudhubs/graal_mvp`

Evaluation and Validation

We built a proof of concept - MicroGraal*:

- Tailored to JavaEE/Spring

We valided our approach on **TrainTicket** v1.0.0:

- Well-established community benchmark
- Focus only on Java microservices – **42** in total

*`https://github.com/cloudhubs/graal_mvp`

Evaluation and Validation

We built a proof of concept - MicroGraal*:

- Tailored to JavaEE/Spring

We valided our approach on **TrainTicket** v1.0.0:

- Well-established community benchmark
- Focus only on Java microservices – **42** in total
- Evaluation performed on 2018 MacBook Pro
  - Analysis Time **15s** per benchmark
  - Average Memory Consumption **850 MB**

*`https://github.com/cloudhubs/graal_mvp`

Evaluation and Validation

We built a proof of concept - MicroGraal*:

- Tailored to JavaEE/Spring

We valided our approach on **TrainTicket** v1.0.0:

- Well-established community benchmark
- Focus only on Java microservices – **42** in total
- Evaluation performed on 2018 MacBook Pro
  - Analysis Time **15s** per benchmark
  - Average Memory Consumption **850 MB**
- The analysis can be done locally

*`https://github.com/cloudhubs/graal_mvp`

Compared with:

- Manual analysis – taken as the ground truth
- Walker et al. 21* using static analysis of source code

Table: Service Dependency Graph Data Analysis

Numbers/Approaches	Manual	Source	Bytecode
REST Calls	146	146	146
Endpoints	261	261	261
Request Pairs in SDG	142	114	123
Links in SDG	90	82	82

*A. Walker, I. Laird, and T. Cerny, "On automatic software architecture reconstruction of microservice applications," in Information Science and Applications. Singapore: Springer Singapore, 2021, pp. 223–234.

Compared with:

- Manual analysis – taken as the ground truth
- Walker et al. 21* using static analysis of source code

Table: TrainTicket: Context Map Data Analysis

Numbers/Approaches	Manual	Source	Bytecode
Entity Bounded Context	117	108	116
Relation Bounded Context	43	39	43
Entity Context Map	84	76	84
Relation Context Map	24	20	24

*A. Walker, I. Laird, and T. Cerny, "On automatic software architecture reconstruction of microservice applications," in Information Science and Applications. Singapore: Springer Singapore, 2021, pp. 223–234.

Conclusion

- We can analyze compiled Java microservices

Conclusion

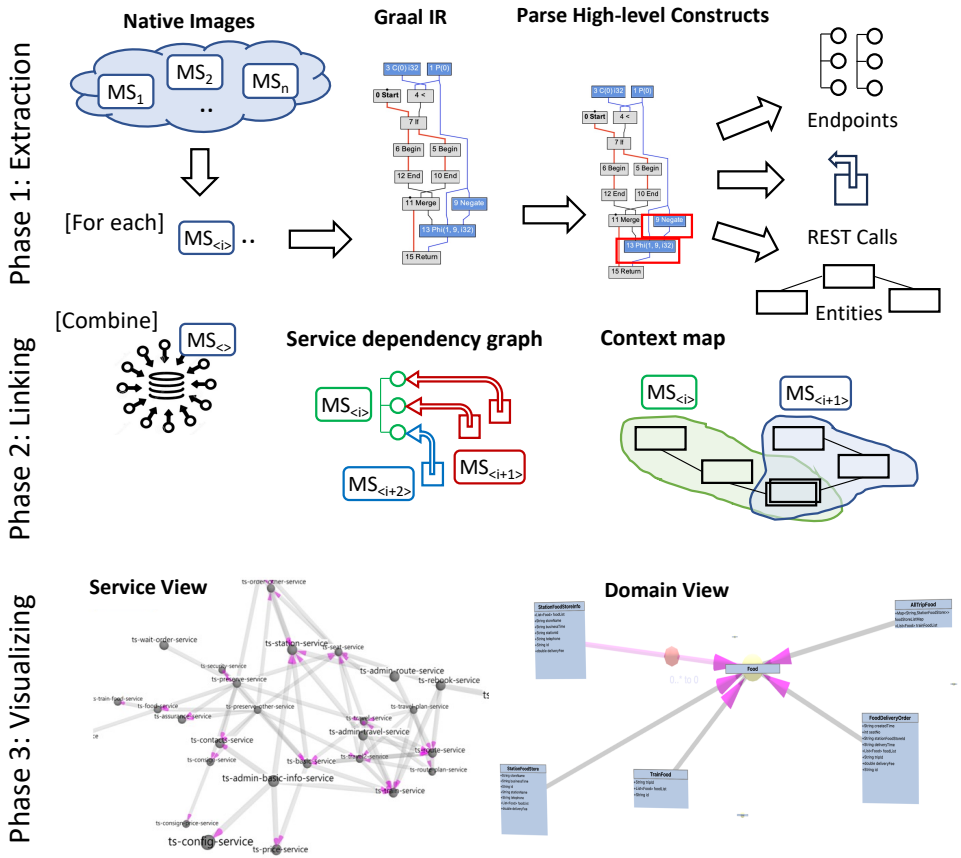
- We can analyze compiled Java microservices
- We can extract documentation, dependencies without running the system

Conclusion

- We can analyze compiled Java microservices
- We can extract documentation, dependencies without running the system
- We can help architects access what is in the system before it goes to production

Conclusion

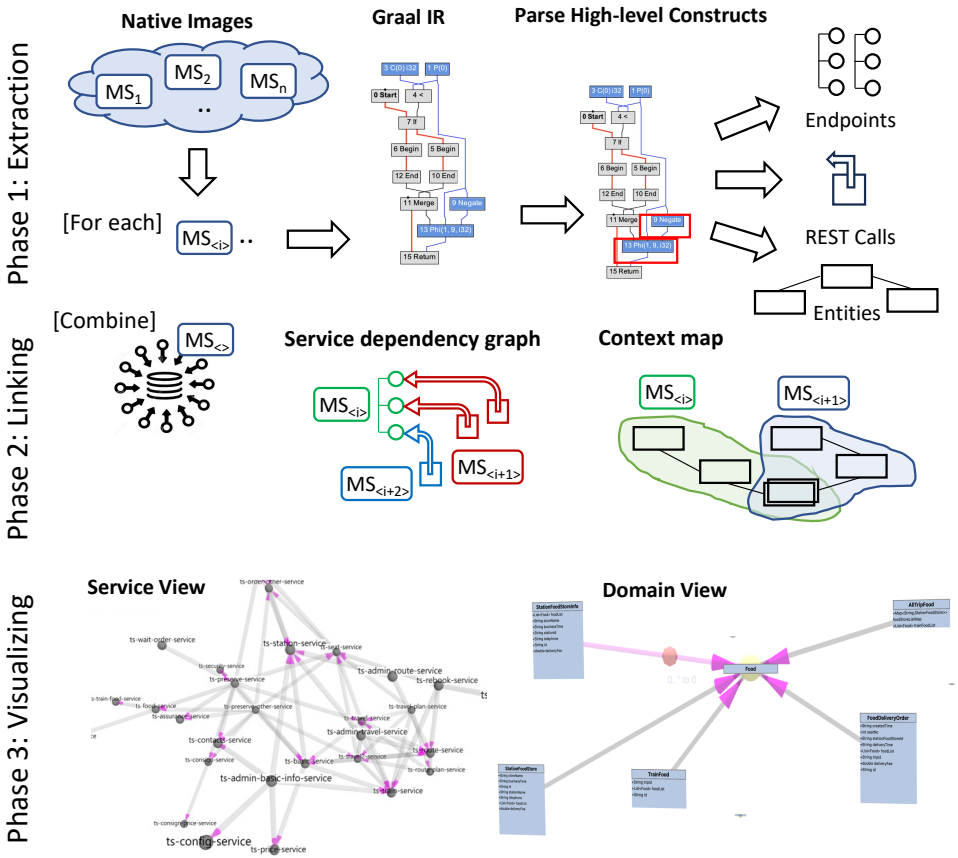
- We can analyze compiled Java microservices
- We can extract documentation, dependencies without running the system
- We can help architects access what is in the system before it goes to production



- Three-step Methodology

Conclusion

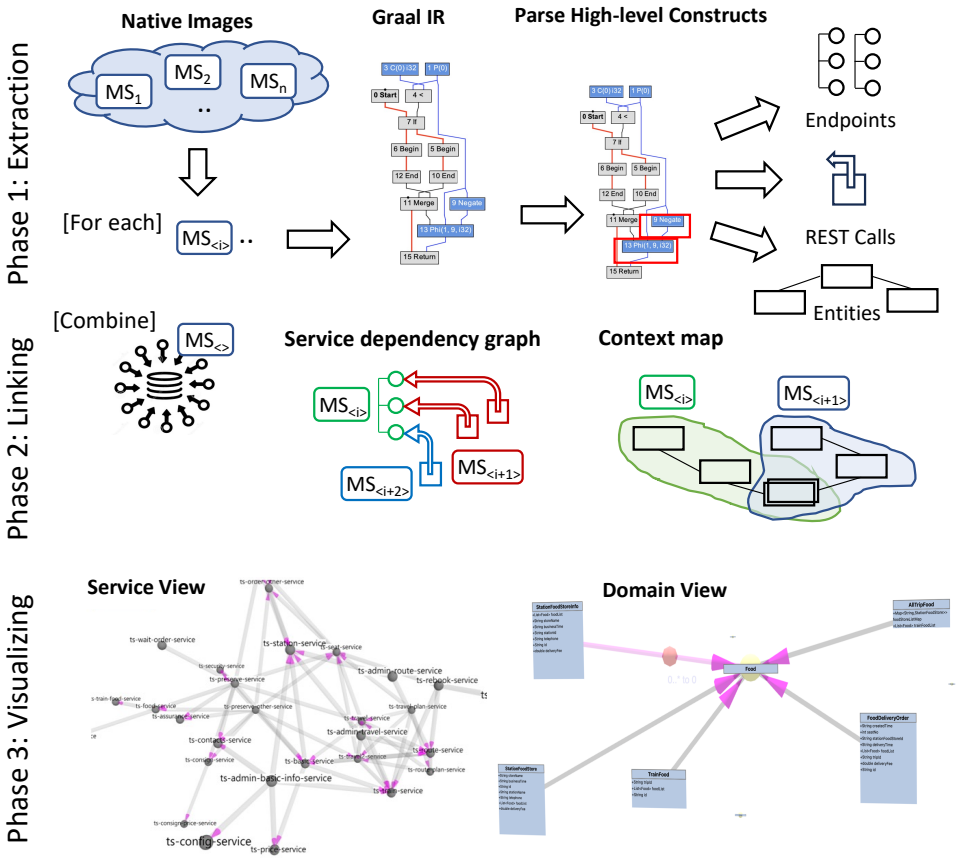
- We can analyze compiled Java microservices
- We can extract documentation, dependencies without running the system
- We can help architects access what is in the system before it goes to production



- Three-step Methodology
- Future work:
  - Polyglot systems
  - Advanced static analyses (taint, data flow)

Conclusion

- We can analyze compiled Java microservices
- We can extract documentation, dependencies without running the system
- We can help architects access what is in the system before it goes to production



- Three-step Methodology
- Future work:
  - Polyglot systems
  - Advanced static analyses (taint, data flow)
- Contact:
  - ikozak@fit.vutbr.cz
  - d-kozak.github.io